LEVERAGING SIDE-CHANNEL SIGNALS FOR SECURITY AND TRUST

A Dissertation
Presented to
The Academic Faculty

By

Nader Sehatbakhsh

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

August 2020

LEVERAGING SIDE-CHANNEL SIGNALS FOR SECURITY AND TRUST

Approved by:

Dr. Milos Prvulovic, Advisor
School of Computer Sicence
Georgia Institute of Technology

Dr. Alenka Zajic, Co-Advisor
School of Electrical Engineering
Georgia Institute of Technology

Dr. Moinuddin K.Qureshi
School of Electrical Engineering
Georgia Institute of Technology

Dr. Angelos Keromytis
School of Electrical Engineering
Georgia Institute of Technology

Dr. Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Date Approved: May 12, 2020

I dedicate this thesis to my parents Nahid and Hooshang Sehatbakhsh, my sister Negar Sehatbakhsh, and to my wife Elham Bagherisereshki. I wouldn't be here without your love and support.

This thesis is also dedicated to my late undergraduate advisor Mehdi Fakhraie, who inspired me how to be a better person.

ACKNOWLEDGEMENTS

I would like to sincerely thank my advisors, Dr. Alenka Zajic and Dr. Milos Prvulovic, for giving me an opportunity to work on this interesting topic. Their time, ideas, feedback, and support made the completion of this thesis possible. Beyond this thesis, I am highly grateful for their time and guidance in helping me to become an independent researcher and an effective advisor. I will carry these influences to my career in the future.

I also would like to thank my thesis committee: Dr. Angelos Keromytis, Dr. Moinuddin K. Qureshi, and Dr. Hyesoon Kim. Their time, support, and inputs were essential in improving this thesis.

The formation of this thesis could not be possible without teamwork and help of my brilliant collaborators and lab-mates including Dr. Alireza Nazari, Moumita Dey, Dr. Baki Yilmaz, Dr. Monjur Alam, Haider Khan, Dr. Robert Callan, Dr. Chia-lin Cheng, Dr. Sunjae Park, Dr. Ngoc Loung Nguyen, Dr. Prateek Juyal, and Frank Werner.

Finally, I owe the completion of this thesis to the help of many of my dear friends who have fulfilled a critical role in my life and supported me in difficult times. Specifically I would like to sincerely thank three people whom together we grew up, laughed, and cried: Shahrad, Amir (Khan), and Siros.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Side-channel signals are referred to any analog-domain or digital-domain *by-product* of computation on electronic devices. They act as additional, and often unwanted, sources of information about the device and its activities. With the proliferation of computing systems in our world, from servers to internet-of-things devices, side-channel signals have become significantly more available and accessible to measure and leverage. This availability provides both opportunities and security threats. On one hand, these side-channels may "leak" sensitive information about the system, and if exploited by an adversary, it would pose security threats to the system. On the other hand, however, these signals can be leveraged as extra sources of information that can be used for benign and useful purposes such as debugging/profiling, malware/intrusion detection, and even establishing trust.

Given these opportunities and threats, understanding how these side-channels are created and developing frameworks to **(a)** *discover, model, and mitigate side-channel signals on modern systems*, and **(b)** *leverage side-channel signals for useful purposes* (e.g., profiling, intrusion detection, establishing trust) to improve the security and/or performance of resource-constrained devices have become an important and active area of research.

To address these challenges, this thesis addresses both possible use-cases of (analog) side-channels: *leveraging* side-channels for benign applications (Chapters 1-6), and *exploiting* (and mitigating) side-channels for information leakage (Chapters 7-8).

Specifically, this thesis develops methods and frameworks to identify, model, and *leverage* side-channel signals particularly analog-domain electromagnetic (EM) emanations by synthesizing techniques from the fields of computer architecture, security, signal processing, electromagnetics, machine learning, and software engineering.

We carefully analyze EM side-channel signals from a broad range of embedded devices and cyber-physical-systems, and demonstrate that *they can be utilized for a variety of useful applications* namely profiling (Chapter 3), intrusion detection (Chapter 4-5), and establish-

ing trust through software attestation (Chapter 6) based on a novel method called *Spectral Profiling*. Further, this thesis also investigates new methods for *identifying and modeling* analog side-channel signals to create future secure and side-channel resistance systems. We first discover a new side-channel vulnerability on modern computers (Chapter 7). Further, a novel method to effectively and accurately model analog-domain side-channel signals is presented (Chapter 8).

Based on these studies, my **thesis statement** is that *"analog-domain side-channel signals can be leveraged for security and establishing trust, and can be accurately modeled on a variety of devices and under different sources of variations."*

CHAPTER 1
INTRODUCTION

Computers are becoming increasingly pervasive and critical in our lives due to the rise of embedded devices and cyber-physical systems. While these systems are performing their primary task: computing, they create unwanted footprints that can leak potentially sensitive information about the system through unconventional channels called ***side-channels***. With the ever-increasing reliance of our lives to computing systems, it is important, more than ever, to deeply understand how these side-channel signals can be created, how side-channel leakage can be mitigated and even leveraged for useful applications, and also, how future systems should be designed to be robust against side-channel attacks?

To address these challenges, this dissertation is mainly focused on analyzing analog-domain side-channels and proposing new methods to:

- leverage side-channel signals for useful purposes (e.g., profiling, intrusion detection, establishing trust) to improve the security and/or performance of resource-constrained devices such as embedded and cyber-physical systems;
- discover, model, and mitigate side-channel signals on modern systems.

1.1 Leveraging Side-Channels for Useful Purposes

Using this fact that analog side-channels are correlated with data and application being processed in a device (e.g., an IoT), this thesis develops methods and frameworks for *leveraging* side-channel signals (particularly electromagnetic emanations) for improving the performance and security of resource-constrained systems (e.g., IoTs, CPSs, etc.). Followings briefly describe the methods/frameworks that are developed in this work:

1.1.1 Spectral Profiling: Observer-Effect-Free Profiling by Monitoring EM Emanations

We begin our studies by presenting a novel method on analyzing software activities using EM side-channel signals unintentionally emanated by the target device. We then demonstrate how this method can be used for profiling and code-tracking using our framework, *Spectral Profiling*, an observer-effect-free profiler which leverages EM emanations to track the code and determine which parts of the program have executed at what time without instrumenting or otherwise affecting the profiled system. We will show that this framework can be used for improving the performance or debugging purposes and is particularly attractive for resource-constrained systems with limited processing power. Our results confirm that *Spectral Profiling* yields useful information about the runtime behavior of a program, allowing it to be used for profiling in systems where profiling infrastructure is not available, or where profiling overheads may perturb the results too much ("Observer's Effect").

1.1.2 EDDIE: EM-based Detection of Deviations in Program Execution

Building upon the idea of using EM emanations for profiling, we make an additional observation that EM signals can also be used to security monitor devices for intrusion/malware detection. To demonstrate the feasibility of this idea, we develop our novel *intrusion detection* framework, *EDDIE*, which constantly monitors the target system via analyzing the EM emanations for detecting anomalies in program execution, such as malware and other code injections, without introducing any overheads, adding any hardware support, changing any software, or using any resources on the monitored system itself. Since *EDDIE* requires no resources on the monitored machine and no changes to the monitored software, it is especially well-suited for security monitoring of embedded and IoT devices. We evaluate *EDDIE* on a real IoT system and in a cycle-accurate simulator, and find that even relatively brief injected bursts of activity are detected by *EDDIE* with high accuracy.

1.1.3 REMOTE: Robust External Malware Detection Framework by Using Electromagnetic Signals

To further improve the robustness and applicability of our intrusion detection framework, *EDDIE*, and addressing a number of practical issues such as different sources of variability, we develop a new robust intrusion detection framework called *REMOTE* by introducing a new distance metric and comparison model (for comparing the spikes between the model and the monitored device) and a new training method which eliminates the need for instrumentation. Similar to *EDDIE*, this framework detects malware/intrusion by externally observing EM signals emitted by Cyber-Physical System (CPS) while running a known application, in real-time and with a low detection latency, and without any a priori knowledge of the malware. *REMOTE* does not require any resources or infrastructure on, or any modifications to, the monitored system itself, which makes it especially suitable for malware detection on CPS where hardware and energy resources may be limited.

To demonstrate the usability of *REMOTE* in real-world scenarios, we port two real-world programs (an embedded medical device and an industrial PID controller), each with a meaningful attack (a code-reuse and a code-injection attack), to four different hardware platforms. We also port shellcode-based DDoS and Ransomware attacks to five different standard applications on an embedded system. To further demonstrate the applicability of *REMOTE* to commercial CPS, we use it to monitor a Robotic Arm. Our results on all these different hardware platforms show that, for all attacks on each of the platforms, *REMOTE* successfully detects each instance of an attack and has $<0.1\%$ false positives. We also systematically evaluate the robustness of *REMOTE* to interrupts and other system activity, to signal variation among different physical instances of the same device design, to changes over time, and to plastic enclosures and nearby electronic devices. This evaluation includes hundreds of measurements and shows that REMOTE achieves excellent accuracy.

1.1.4 EMMA: Hardware/Software Attestation Framework for Embedded Systems Using Electromagnetic Signals

As yet another application for side-channel signals, we propose using physical side-channel signals for other helpful applications such as *establishing trust* and *software attestation*. Establishing trust for an execution environment relies on attestation, where a potentially untrusted system (prover) computes a response to a challenge sent by the trusted system (verifier). The response computation typically involves measurement (e.g., a checksum) of the prover's program code and execution environment, which the verifier checks against expected values for a "clean" (trustworthy) system. The main challenge in attestation is that, in addition to checking the response, the verifier also needs to verify the integrity of the response computation itself, i.e., that response computation itself has not been tampered with to produce expected values without measuring the verifier's actual code and environment. On higher-end processors, this integrity of response computation is verified cryptographically, using dedicated trusted hardware (e.g., SGX). On embedded systems, however, form factor, battery life, and other constraints prevent the use of such sophisticated hardware support. Instead, a popular approach is to use the request-to-response time as a way to establish some level of confidence about the integrity of the response computation itself. However, the overall request-to-response time provides only one coarse-grained measurement from which the integrity of the attestation is to be inferred, and even this one measurement is noisy because it includes the round-trip network latency and/or variations due to micro-architectural events. Thus, the attestation is vulnerable to attacks where the adversary has tampered with response computation, but the resulting additional computation time is small relative to the overall request-to-response time.

To tackle this problem, we make a key observation that the existing approach of execution-time measurement for attestation is only one example of using externally measurable side-channel information and that other side channels, some of which can provide much finer-grain information about the response computation, can be used. As a proof of concept, we

propose *EMMA*, a novel method for attestation that leverages EM side-channel signals that are emanated by the system during response computation, to confirm that the embedded device has, upon receiving the challenge, actually computed the response using the valid program code for that computation. This new approach requires physical proximity, but imposes no overhead to the system, and provides highly accurate monitoring during the attestation process. We implement *EMMA* on a popular embedded system, Arduino UNO, and evaluate our system with a wide range of attacks on attestation integrity, and show that *EMMA* can successfully detect each instance of these attacks.

1.2 Designing Side-Channel Resistance Systems

While the above frameworks demonstrate the benefit of understanding side-channels and their usefulness for benign applications (e.g., security, profiling, etc.), it is important to mention that *proper analysis and modeling of side-channels can also be helpful to improve security of the system and reducing information leakages*. Given the growing importance of designing side-channel resistance systems, the last two chapters of this thesis is focused on designing secure and side-channel resistance processors by *(a)* discovering a new side-channel vulnerability on modern computers, and *(b)* proposing a novel simulation tool for estimating analog side-channel leakage for a any given software/application. Followings describe them in more details:

1.2.1 A New Side-Channel Vulnerability on Modern Computers by Exploiting Electromagnetic Emanations from the Power Management Unit

In Chapter 7, we present a new *micro-architectural* vulnerability, which is created by power management units of modern computers and can be exploited through electromagnetic, and potentially other, side-channels. The key observations that enable us to discover this side-channel are: 1) in an effort to manage and minimize power consumption, modern micro-processors have a number of possible operating modes (*power states*), in which various

sub-systems of the processor are powered down, 2) for some of the transitions between power states, the processor also changes the operating mode of the voltage regulator module (VRM) that supplies power to the affected sub-system, and 3) the EM emanations from the VRM are heavily dependent on its operating mode. As a result, these state-dependent EM emanations create a side-channel that can reveal which programs are currently executing, and potentially other sensitive information about the executed programs.

To demonstrate the feasibility of exploiting this vulnerability, we create a *covert channel* that uses changes in the processor's power states to *exfiltrate* sensitive information from a system that is otherwise secured and completely isolated (*air-gapped*), and then receives that information using a compact, inexpensive receiver placed in proximity to the system. To demonstrate the *severity* of this vulnerability, we also show that information can be successfully exfiltrated even if the receiver is *several meters* away from the system, and even if the system and the receiver are separated by a wall. Compared to the state-of-the-art, the proposed covert channel has >3x higher bit-rate. Finally, to demonstrate that this new vulnerability is not limited to being used as a covert channel, we demonstrate how it can be used for attacks such as *keystroke logging*.

1.2.2 EMSim: A Microarchitecture-Level Simulation Tool for Modeling Electromagnetic Side-Channel Signals

Side-channel attacks have become a serious security concern for computing systems, especially for embedded devices, where the device is often located in, or in proximity to, a public place, and yet the system contains sensitive information. To design systems that are highly resilient to such attacks, an accurate and efficient design-stage quantitative analysis of side-channel leakage is needed. For many systems properties (e.g., performance, power, etc.), cycle-accurate simulation can provide such an efficient-yet-accurate design-stage estimate. Unfortunately, for an important class of side-channels, electromagnetic emanations, such a model does not exist, and there has not even been much quantitative evidence about

what level of modeling detail would be needed for high accuracy.

In Chapter 8, we present *EMSim*, to simulate EM side-channel signals cycle-by-cycle using a detailed micro-architectural model of the device. To evaluate *EMSim*, we compare its signals against actual EM signals emanated from real hardware (FPGA-based RISC-V processor), and find that they match very closely. To gain further insights, we also experimentally identify how the accuracy of the simulation degrades when key micro-architectural features (e.g., pipeline stall, cache-miss, etc.) and other hardware behaviors (e.g., data-dependent switching activity) are omitted from the simulation model. We further evaluate how robust the simulation-based results are, by comparing them to real signals collected in different conditions (manufacturing, distance, etc.). Finally, to show the applicability of *EMSim*, we demonstrate how it can be used to measure side-channel leakage through simulation at design-stage.

1.3 Dissertation Outline

The remainder of this thesis is organized as follows: In Chapter 2, we provide a brief background on side-channels, and then describe on our novel method for analyzing software activities through AM-modulated frequency-domain signals using EM side-channel signals which we call *Spectral Profiling*. We then demonstrate how this method can be used for profiling and debugging, and present its results in Chapter 3. Next, we propose our novel method, *EDDIE*, for intrusion detection using EM signals (Chapter 4). In Chapter 5, we discuss the shortcomings of *EDDIE* and propose a robust framework, *REMOTE*, that can address and completely solve these issues. In Chapter 6, we present another use-case of EM side-channel signals by demonstrating how EM emanations can be used for establishing trust and software attestation. In Chapter 7, we begin our analysis on discovering and modeling analog side-channels. We present a new side-channel vulnerability on modern laptops. Chapter 8 presents our novel approach in modeling analog side-channels. Finally, Chapter 9 summarizes this thesis and provide some future directions to this work.

CHAPTER 2

SPECTRAL PROFILING: OBSERVER-EFFECT-FREE MONITORING BY LEVERAGING ANALOG SIDE-CHANNEL SIGNALS

2.1 Motivation

Cyber-Physical Systems (CPS) are proliferating in numbers and importance. By 2025, CPS is expected to be a USD 6.2 trillion market globally (this is 8% of the entire world's 2016 GDP), and most of that is expected to be in healthcare (USD 2.5 trillion) and manufacturing (USD 2.3 trillion) [1, 2]. While the CPS world can provide many benefits for industries and individuals, it, unfortunately, comes with new opportunities for cyber-attacks. By 2020, it is estimated that more than 25% of known attacks in enterprises will involve the CPS while less than 10% of IT security spending will be on CPS security, indicating that there is an emerging need for more attention on CPS security [3].

There is a wide range of CPS security targets: cameras, cars, industrial PLCs, critical infrastructures such as the power grid (power distribution, nuclear and other power-plants, etc.), hospitals and embedded medical devices, etc. Many of these targets have already been attacked (e.g., DDoS attacks [4] in DNS services occurred by Mirai malware-infected CPS, etc.). Figure 2.1 shows a brief overview of recent attacks on cyber-physical-systems.

Because CPSs use various and customized hardware and software, they may not be upgraded or updated as often as general-purpose systems, and software updates are even less frequent for devices where extensive verification or regulatory approval is needed. This makes cyber-physical systems challenging to keep up-to-date with the ever-evolving landscape of possible vulnerabilities and threats [5]. Furthermore, existing techniques for intrusion detection, such as those based on scanning for malware signatures [6], sandboxing [7], hardware support [8, 9, 10], machine learning [11], and dynamic analysis [12],

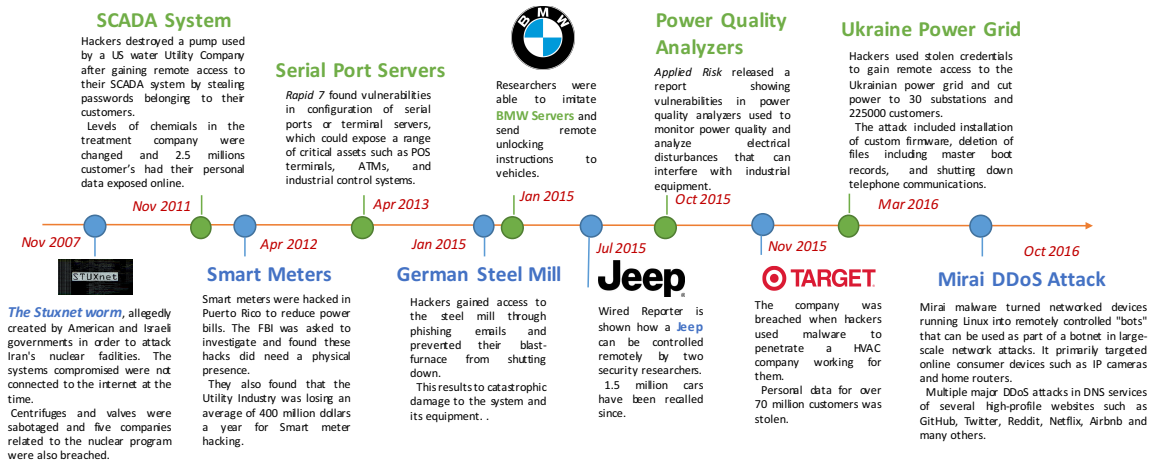


Figure 2.1: Timeline of recent cyber-attacks and exploits on IoT systems.

impose significant computational overhead and/or increase hardware cost, so they are difficult to adapt to CPSs that often have severe performance, resource, power, and cost constraints. Moreover, the low-complexity nature of CPS makes it easier to obtain full access to the device and then disable or even coopt their monitoring functionality. Further, existing *instrumentation-based* approaches for profiling and/or security monitoring [13, 14, 15, 16, 17] incurs significant overhead to the system and/or significantly impact the *actual* performance of the application (i.e., "observer-effect").

To address these challenges, an alternative approach for intrusion detection and/or profiling is to use side-channel signals as the source of the information for building a "reference" model of the device and its applications and leverage that model and the side-channel signal(s) for monitoring and/or debugging the system. Such an approach has zero-overhead on the system and completely eliminates the observer-effect while it provides an extra layer of the security to the system.

Using insights mentioned above, this thesis develops methods and frameworks for leveraging side-channel signals (particularly electromagnetic emanations) for improving the performance and security of resource-constrained systems (e.g., IoTs, CPSs, etc.). Followings provide a brief background on analog-domain side-channels, and then describe the

main idea proposed in this thesis for side-channel monitoring.

2.2 Unintentional Amplitude-Modulated Side-Channel Signals

2.2.1 Background

Side-Channel Signals

These signals are unintentionally generated as an artifact during computation. Side-channels can be classified into two main categories: *Digital/Micro-Architectural* and *Analog/Physical*. Digital side-channels typically rely on *shared hardware* resources in the computer. Examples of these channels include caches [18, 19, 20, 21, 22, 23, 24], micro-architectural units [25, 26], DRAM and memory bus [27, 28], processor frequency settings [29], branch predictors [30, 31, 32], GPUs [33], TLBs [34], etc. Physical side-channels, however, rely on physical characteristics of the system such as EM emanations [35, 36, 37, 38, 39, 40, 41, 42], variation in power consumption [43, 44, 45, 46, 47], sound/acoustic [48, 49, 50, 51, 52, 53], temperature [54, 55, 56], chassis potential variation [57], crosstalk [58], peripherals [59, 60], etc.

The main difference between the two categories is that the former (digital), typically, can be measured *within* the system (i.e., by another process), while the latter often requires some physical proximity to the device for measurements or, alternatively, needs to access some sensors on the board to read the desired value. Due to this limitation, physical side-channel attacks often have much lower transmission rate. However, unlike digital side-channels, physical side-channels are much more challenging to mitigate, and are often too expensive and impractical to eliminate. This, makes physical side-channels particularly attractive in scenarios where the system is well-protected from digital side-channels through isolation and/or partitioning [61]. For example, a popular method for strong isolation is creating an *air-gap*, where all the computer's connections to the outside world are either disabled or monitored. However, data can still be exfiltrated using a physical side-channel.

Electromagnetic Side-Channel Signals

It has been well documented in the literature that electronic circuits within computers generate detectable EM emanations called side-channel EM radiation [62, 63, 64]. The existence of side-channel EM radiation, and the potential risk it poses for computer security, was reported in the open literature as early as 1966 [62, 65], and much of this literature refers to the even older (classified) TEMPEST work [65, 66]. Much of the early work on EM emanations focused on information leakage created by signals from cathode-ray-tube (CRT) computer monitors [67] and on how to reduce such risks [66]. In practice, however, these risks were largely eliminated by the demise of CRT monitors, as their successors, LCD monitors, create much weaker EM fields.

Research interest in compromising EM emanations has been renewed with the mass-market introduction of smartcards (e.g., EMV "chip" credit/debit cards). A typical smartcard has a microcontroller operating at low frequencies (<300 MHz) and usually executes a single program (cryptographic authentication). EM emanations resulting from their program activity can leak information about the embedded cryptographic key(s) [63, 68], both through direct emanations that are caused by intended current flows within circuits (from switching activity while adding two numbers in a processor) and through indirect emanations caused by electromagnetic coupling among chip circuits. Numerous countermeasures have been proposed that reduce information leakage from smartcards [66, 69, 70, 71, 72, 73, 74, 75, 76, 77], including adding low-cost shielding (e.g., metal foil), using asynchronous circuits, and changing the layout of circuitry.

Side-Channel attacks on high-performance (server, desktop, and laptop) systems are more difficult because they often require capturing signals at a sampling-rate much faster than the devices' clock rate, which is impractical for GHz clocks [64, 78, 79]. Despite these difficulties, it has been shown that information can be transmitted via EM emanations [80], even in the presence of significant countermeasures [64], and cryptographic keys can be extracted from modern computers using EM side-channel analysis [78].

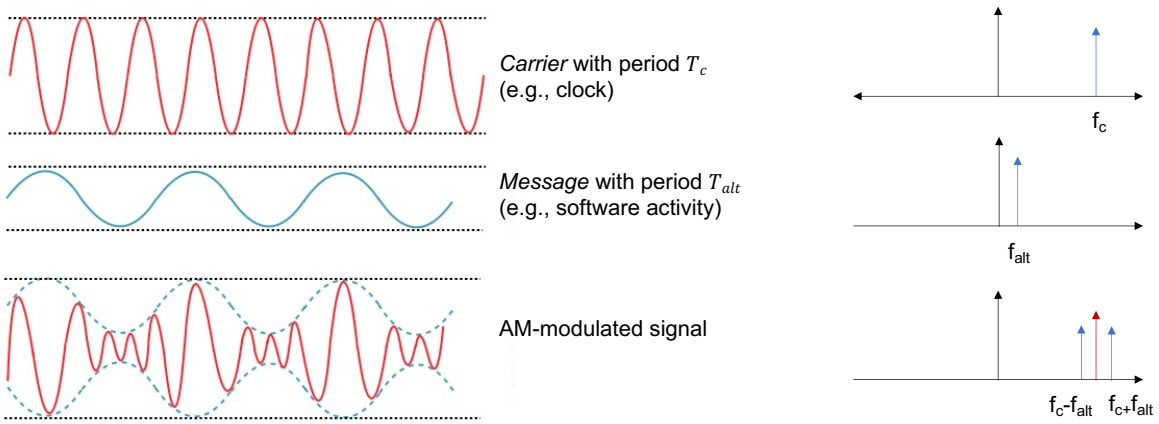


Figure 2.2: A Sinusoidal carrier modulated by a sinusoidal signal shown in time-domain (left) and frequency-domain (right).

Given these findings, it is only natural to wonder whether we can learn more about a program's behavior by observing side-channel signals. Recently, for instance, current (power) fluctuations were used to identify webpages during browsing [81] and even find anomalies in software activity [82, 83]. Results in existing works [79, 84, 85, 86] show that differences between different instructions can be measured in EM analog signals across different devices (e.g., desktops, laptops, FPGAs) and also identify which aspects of program activity modulate which EM-emanated signals.

2.2.2 What is Spectral Profiling?

Among all emanated signals from a real-time system, some of the strongest and farthest propagating signals are created when an existing strong periodic signal (e.g., a clock signal) and its side-bands become stronger or weaker (amplitude-modulated) depending on the system's activity. These "accidental" data transmissions can propagate potentially sensitive information about the system (and its applications) with high Signal-to-Noise Ratio (SNR) thus it provides an opportunity to leverage (EM) side-channel signals more effectively (either for a benign usage such as profiling and/or malicious usage such as information leakage).

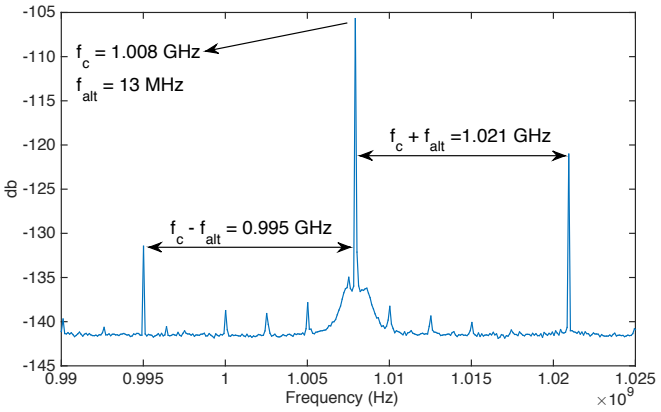
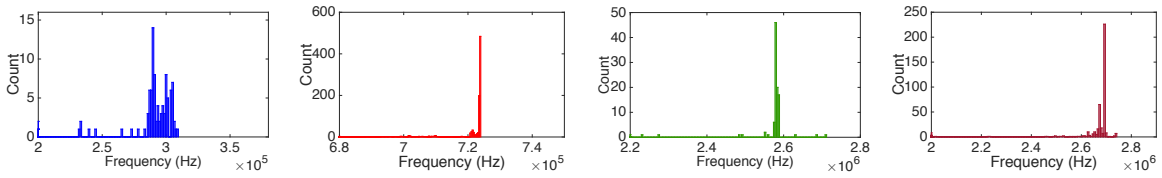


Figure 2.3: Spectrum of an AM modulated loop activity..

To better understand this phenomenon, Figure 2.2 shows the time-domain and spectrum of an ideal carrier signal (at frequency f_c) that is modulated by an ideal sinusoidal signal at frequency f_{alt}. In addition to the carrier signal, this spectrum has strong "side-band" signals offset by f_{alt} , i.e., at frequencies $f_c - f_{alt}$ and $f_c + f_{alt}$. This would be the spectral pattern to look for when a periodic signal has a perfectly stable frequency and is modulated by a pattern of activity with a fixed period of $T_{alt} = 1/f_{alt}$ with no variation in timing.

In reality, both the carrier (e.g., square-wave shaped clock signal) and the periodic activity (e.g., loops in a program) are slightly unstable over time which causes variations in the periodicity and hence the spectrum. An example of a "real" unintentional AM-modulation signal can be seen in Figure 2.3 where a device with clock frequency, $f_c = 1.008$GHz, and a loop with frequency, $f_{alt} = 13$MHz, is shown. As can be seen in the figure, the carrier is slightly *spread* around its nominal value and this spreading is also present (i.e., "inherited") in the two side-band signals. The carrier is spread out mainly due to the (intentional or unintentional) non-ideal properties of computing system (e.g., spread-spectrum clocking, imprecise timing, etc.) and existing circuit/environmental noises.

Apart from the carrier, the side-bands (i.e., the program activity) can be also spread-out due to the variations in the *per-iteration* execution time of the loop (mainly due to the existence of different execution paths in the code). Figure 2.4 shows this variation for



(a) (b) (c) (d)

Figure 2.4: Histogram of per-iteration execution time for four different loops in a standard benchmark application called *basicmath* [87].

four different loops in an application (called *basicmath*) from an standard benchmark suite, MiBench [87]. As shown in the figure, while some loops have quite stable per-iteration time, e.g., loop(b), others may have some variations, e.g., loop(a), which, in turn, causes a spread-out signature in the spectrum.

Using this observation, by knowing the frequency of each path before runtime, during execution we can deduce (1) which path in the program is currently active, (2) how much time program has spent in this path, and (3) how many times this path has been executed. Further, as we will show later, this information can be used for intrusion detection, e.g., a change in the shape of each phase/loop indicates a possible anomaly in its execution (we will discuss this in more details in Chapter 4).

As an application goes through different functions and loops, its spectrum is expected to change over time. To capture this dynamic spectrum behavior of an application, we use a short-time Fourier transform (STFT) that is defined as [88]

$$STFT\{s(t)\}(\tau,\omega) \equiv X(\tau,\omega) = \int_{-\infty}^{+\infty} \mathrm{x(t)w(t-\tau)e^{-j\omega t}}dt, \tag{2.1}$$

and then compute a spectrogram as

$$spectrogram(t,w) = |STFT(t,w)|^2. \tag{2.2}$$

In STFT, a long signal is divided into shorter, equal, and slightly overlapping segments (windows). Computing Fourier Transform of these segments separately and plotting vari-

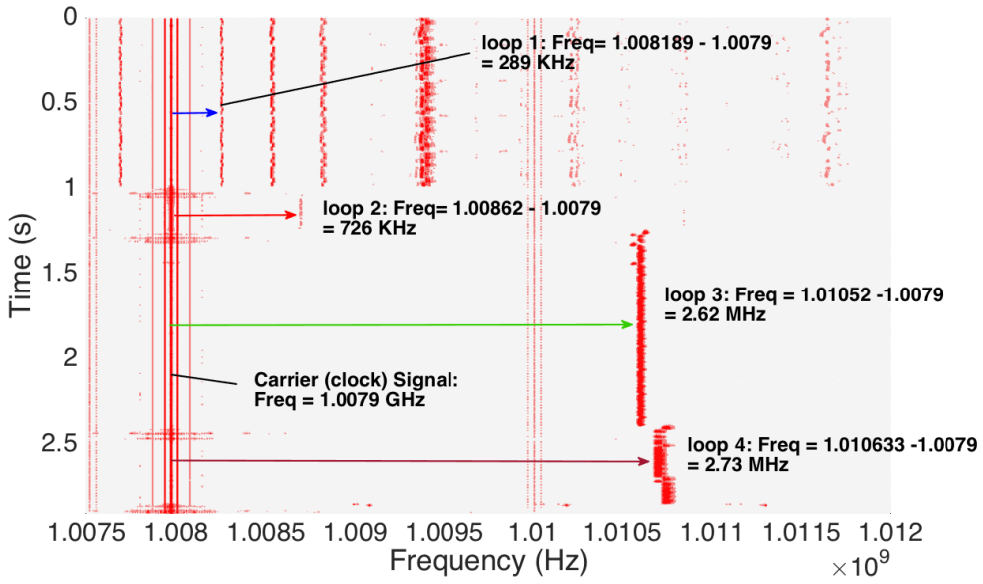


Figure 2.5: Spectrogram (spectrum over time) of the *basicmath* application with four different loops.

able spectra over time allows us to capture local spectrum characteristics. Figure 2.5 shows the spectrogram of the *basicmath* [87] application. The bold line at 1.008MHz is the clock signal. The periodic program behavior amplitude-modulates this signal, and the straight lines to the right of this line represent the upper sideband of the modulated signal, i.e., they have the spectrum that corresponds to program behavior but that spectrum is shifted upward in frequency by the clock frequency. In this execution the four loops are executed one after the other (shown by arrows). The per-iteration execution time each loop is also shown in Figure 2.4. Comparing these two figures, it can be clearly seen that loops with more spread-out per-iteration time, e.g., loop(a), have thicker line (i.e., more frequency variation) in the spectrogram.

Leveraging these set of observations and using STFT, we can then develop a framework to use these AM-modulated side-channel signals for software profiling and/or other helpful applications. *To the best of our knowledge, this is the first time that these unintentional EM emanations are used for such helpful purposes. In the next chapters, we describe how such a method can be used for a variety of helpful purposes.*

CHAPTER 3

DESIGNING A MONITORING FRAMEWORK USING SPECTRAL PROFILING

3.1 Abstract

In this Chapter, we describe how the concept of *Spectral Profiling* can be used for profiling and code-tracking. This framework is an observer-effect-free profiler which leverages EM emanations to track the code and determine which parts of the program have executed at what time without instrumenting or otherwise affecting the profiled system. We will show that this framework can be used for improving the performance or debugging purposes and is particularly attractive for resource-constrained systems with limited processing power. Our results confirm that our Spectral Profiling framework yields useful information about the runtime behavior of a program, allowing it to be used for profiling in systems where profiling infrastructure is not available, or where profiling overheads may perturb the results too much ("Observer's Effect").

The main contributions of this work are:

- A new approach for profiling that requires no profiling-related support or activity on the profiled system.
- A proof-of-concept implementation of this approach to demonstrate its feasibility in practice.
- An experimental evaluation that shows our new approach achieves high profiling accuracy, both in a real system and in cycle-accurate simulation.

3.2 Design Overview

Spectral Profiling monitoring framework has two phases, training and profiling. In the training, we run the application with known training inputs to identify which spectra correspond to which part of the program (mostly loops), and also to identify the valid orderings between the parts of the program. In the profiling, we run the application with unknown inputs, record how the spectrum change over time, and combine that with the information from training to detect which part of the program is executing at each point in time.

Spectral Profiling's recognition of activity is based on recognizing the corresponding spectrum. Any spectrum fundamentally corresponds to the signal observed over some interval of time (window), and the duration of this window represents a trade-off between temporal resolution and frequency resolution. Temporal resolution corresponds to being able to tell where exactly some program activity begins and ends. Fundamentally, a spectrum that corresponds to some time window "blurs together" activity for the entire window, so spectra collected with a short window allow more precise identification of the time when program activity has changed. This means that, to improve temporal resolution, we should use spectra collected over very short intervals of time. However, the number of frequency bins (i.e., the frequency resolution) in the spectrum is proportional to the duration of the time interval, so a spectrum collected over a very brief interval "lumps together" similar frequencies into one frequency bin. This means that two program activities that have spectral "spikes" with different shapes and/or similar frequencies cannot be told apart when using short-window spectra because the spectrum only has one bin for the entire frequency range where both spikes are. In our setup, we use a window of 1ms with 75% overlap between consecutive windows, which provides attribution with 0.25 ms granularity and precision between 0.25 ms and 1 ms.

3.3 Training Phase

The goal of the training phase is to collect spectral signatures for all regions of the program, and also to identify the possible/probable sequence of the program's execution, i.e., when one region of code is executed, which regions can possibly be executed immediately after that. To build the training model and collect spectra, we first use instrumentation to measure the average per-iteration execution times for each loop. Then we re-run the program with the same training inputs but without the instrumentation to get the undistorted spectra. Finally, we use the per-iteration execution times and the frequencies of spikes in the spectrum to create spectrum-to-loop mappings.

3.3.1 Finding Per-Iteration Execution Time

We place instrumentation at the beginning and end of the loop body to get timestamps at those points, compute execution time for each iteration, and store it along with information about which loop they correspond to. When this training run ends, we compute the average per-iteration execution time for each loop instance. Note that this per-iteration time is *only* used in training to identify the frequency at which the corresponding spectrum should have a spike: if per-iteration time of a loop is T, we will expect the corresponding spectrum to have a spike at a frequency that is relatively close to $f = 1/T$.

3.3.2 Finding Spectral Signatures for Each Loop

After calculating the frequency of each loop, we re-run the application with same inputs but without any instrumentation or profiling-related activity on the profiled device, and record the spectra for each time window. In each spectrum, we identify the spikes, then compare their frequencies and shapes to the histogram obtained from the previous (instrumented run). The matches are imperfect because instrumentation perturbs the execution time of a loop's iteration, and thus changes the frequency and shape in the histogram. However,

our matching is highly accurate because frequencies that correspond to different loops tend to differ more than the instrumentation-induced errors, because the error introduced by instrumentation is usually in the same direction (increases the per-iteration execution time), and also because our matching approach utilizes the fact that the two runs used the same inputs and thus have the same sequence of loops. For example, in *basicmath* application (cf. Table 3.1 and Figure 2.4 in Chapter 2) phase/loop 3 and phase/loop 4 have relatively similar frequencies, but because we know that phase/loop 3 is likely to have a lower frequency than, and be executed before, phase/loop 4, the spectra corresponding to these phases/loops can still be correctly "assigned". In addition, after successfully assigning spectra to the phases/loops, we will also have the sequences of the "assigned" phases/loops.

Table 3.1 shows the list of frequencies for four loops in *basicmath*. The "measured" column in the table shows the actual frequency of the loop (i.e., in the instrumentation-free run), and the "calculated" column shows the average frequency calculated from the instrumentation-enabled histogram. The relative error between the calculated and measured frequency is up to 2%, but we can still easily match them. Also note that the frequency error introduced by instrumentation increases as the frequency increases. This is because instrumentation has more effect on tight loops.

After matching spectra to loops, we pre-process the (instrumentation-free) spectrum that corresponds to each loop to identify the "spectral signature" for the loop. In our implementation, the signature is a list of frequencies for the strongest spikes in the spectrum, after removing spikes that appear in all spectra (e.g., for EM signals, for example, this eliminates spikes caused by radio stations, etc.). Note that the signature is not just one number

Table 3.1: Measured and Calculated Frequency for loops in *basicmath* application

Loop Number	Frequency (measured)	Frequency (calculated)
Loop 1	289.12 KHz	289.1 KHz
Loop 2	720.3 KHz	721 KHz
Loop 3	2.628 MHz	2.577 MHz
Loop 4	2.733 MHz	2.69 MHz

that corresponds to the fundamental frequency of the loop. Some loops have a group of spikes instead of one spike, because their per-iteration execution time takes several discrete values (with some variation around each of them). In most cases, the spectrum also contains not only the spikes that correspond to the per-iteration execution time (fundamental frequency), but also spikes at multiples (harmonics) of that frequency. These additional spikes help differentiate spectra that correspond to different loops, so the signature we use includes all spikes whose magnitude is sufficiently above the noise floor.

3.4 Profiling Phase

In training, we identified the spectral signature for each loop, and we have also identified the possible/probable sequences of loops (essentially, which loops can execute immediately after which other loops). Profiling consists of running the application with unknown inputs and obtaining profiling information about those runs.

3.4.1 Matching of Loop Spectra

Because the profiling inputs are different from training inputs, it is natural to wonder if the spectrum of a loop will change. We have found that many loops, primarily innermost loops, have spectra that are nearly identical to those found in training. Intuitively, the spectrum changes when the per-iteration execution time changes, and in many loops only the number of iterations changes significantly from input to input, but the work of each iteration (and the statistics of branches and architectural events) remain similar. We call these *Loops with Input-Independent Spectra (LIIS)*, and for these loops the spectrum can be matched to the corresponding spectrum from training.

During profiling, we use the same time window we used during training. For each time window during profiling, we obtain the spectrum for that window, identify the spikes in the spectrum (the spectral signature) and compare that signature to the signatures obtained during training. The comparison is performed by attempting to match the peaks in the

profile-time and training-time signature. For each peak in the profile-time signature, we find the closest peak (according to frequency) in the training-time signature. If that closest frequency differs too much, the peak remains unmatched. If the closest frequency is very similar, the peak is counted as matched. After attempting to match each peak, the number of successfully matched peaks is used as the similarity metric between the signatures.

If the similarity is high between a profile-time spectral signature and the best-matching training-time signature of a loop, we attribute the execution during that profile-time window to that loop. For the vast majority of time-windows that belong to *LIIS* loops, this similarity is very high and the execution is correctly attributed to the correct *LIIS* loop.

However, it is possible that none of the profile-time signatures matches the observed signature well enough. This happens primarily because the spectrum of some loops does change with frequency. For example, a command-line flag may cause every iteration of the loop to take one path in one execution and a significantly different path in another execution or a set of control flows inside the loop that can change the per-iteration execution time of the loop. For these loops, the spectrum still indicates that a loop is executing (spikes in the spectrum) and when the loop begins and ends (spikes appear at one time and disappear later) but the spectrum during profiling no longer matches any of the spectra from training.

3.4.2 Sequence-Based Matching

To attribute execution time to these *Non-LIIS* loops (and report their per-iteration execution time during the profiling run), we rely on the model of possible loop-level sequences constructed during profiling. Sequence-based matching begins after *LIIS* matching is completed for *LIIS* loops. The spectra from time windows that remain unmatched after *LIIS* matching are first clustered according to the same similarity metric we used to match *LIIS* loops to spectra from training, i.e., spectra that have many spikes at similar frequencies will be clustered together. At this point we have clusters where each cluster corresponds to a *Non-LIIS* loop, but we do not yet know which loop in the code this cluster corresponds to.

However, for each "mystery spectrum" we know that it should be matched to a region of code that is not a *LIIS* loop, and the *LIIS* loop spectra observed before and after the "mystery spectrum" tell us which loops have been executed before and after each instance of the "mystery" loop whose cluster we are considering. Fortunately, the model of the application's loop-to-loop transitions restricts the possibilities for matching so that usually only a single *Non-LIIS* loop remains as a possible match. When there are multiple possible matches, i.e., the "mystery spectra" in a cluster could possibly belong to more than one *Non-LIIS* loop, we match the cluster to the *Non-LIIS* loop whose training signature has the highest average similarity to the spectra in the cluster.

3.5 Experimental Results

3.5.1 Experimental Setup

To demonstrate the feasibility and effectiveness of Spectral Profiling, we used it to profile applications running on a single-board computer (A13-OLinuXino), which has a 2-issue in-order ARM Cortex A8 processor with 32kB L1 and 256KB L2 caches, and uses Debian Linux as its operating system (OS). Our Spectral Profiling for this system uses electromagnetic (EM) emanations that are received by a commercial small electric antenna (PBS-E1) [89] that is placed next to the profiled system's processor. The antenna is placed where the clock signal has the strongest Signal-to-Noise ratio (SNR).

A spectrum analyzer (Agilent MXA N9020A) is then used to record the spectra of the signals collected by the antenna. A spectrum analyzer can be relatively costly (several tens of thousands of dollars), but we elected to use a spectrum analyzer primarily because it provides calibrated measurements, and already has support for automating measurements and for saving and analyzing measured results. In additional experiments, we observed similar spectra with less expensive ($<$\$5,000) commercial software-defined radio receivers.

3.5.2 Results

We apply Spectral Profiling to all 13 applications from the automotive, communications, network, and security categories in the MiBnech suite. We used a 1 ms window size with 75% of overlap between windows in all applications.

For training, markers are inserted as described in Section 3.3, except for very tight loops where markers are inserted before and after each loop and, if needed, only iteration-counting is added to the loop. The per-iteration time in this case is computed by dividing the between-markers time by the number of iterations. Each marker reads and records the current clock cycle count from the *ARM Performance Counter Unit* (ARM-PMU) [90], which provides information similar to the x86 "rdtsc" instruction [91]. The training runs are repeated with several different command line flags, in order to identify the sequence of loops that can occur in each application.

The insertion of markers and the identification of possible sequences for an application are both accomplished fully automatically. Identification of loop nests and marker insertion are implemented as a Clang tool, and identification of possible loop sequences in implemented as a pass in LLVM 3.7.

After training, for which we used the small input set [87], we perform actual profiling with the original unmodified code (no markers) and with the large input set [87]. The accuracy we measure is defined as the fraction of execution time for which our method correctly identifies the loop that is currently executing. This accuracy is not 100% because of (1) ***miss-attribution***, during which our algorithm matches the spectrum to a different loop (i.e., loop A is actually executing, but the algorithm matches the spectrum to loop B instead) at loop B, and (2) ***non-attribution***, during which our algorithm finds that the spectrum is too different from loop spectra observed in training, so it leaves such intervals un-attributed. Non-attribution is typically a result of computation whose spectrum varies widely depending on inputs, or activity that has no recognizable spectral signature (e.g., loops whose per-iteration time varies a lot from iteration to iteration).

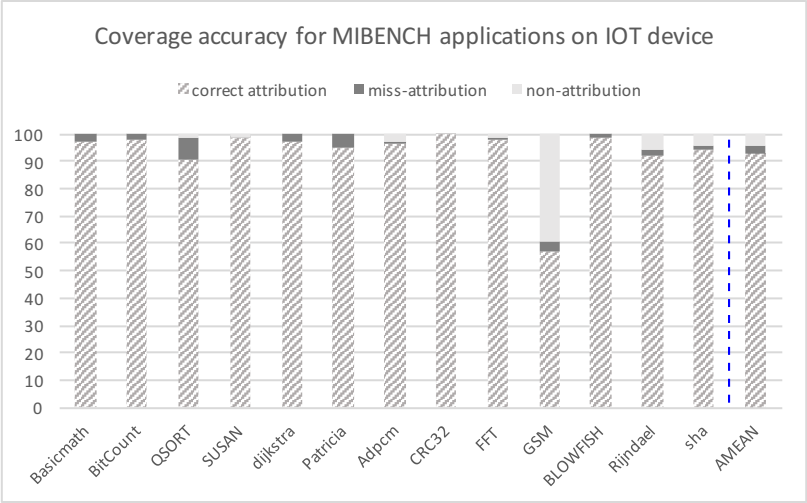


Figure 3.1: Correct attribution (striped portion) as a percentage of the overall profiled execution time.

Figure 3.1 shows the breakdown of profiled execution time into time that was accurately attributed, time that was miss-attributed, and non-attributed time. In all benchmarks except GSM, our method provides correct attribution during at least 90% of the execution time, with the arithmetic mean at 93%. Miss-attribution occurs during less than 4% of the execution time, except in QSORT, where miss-attribution occurs during 8% of the time. The larger miss-attribution for QSORT occurs primarily because the `std::qsort` library function does not have a stable signature, so it is often miss-attributed. It is possible that the variation in `std::qsort` spectra is a result of having multiple loops in that function. Unfortunately, our marker insertion did not include library code so our scheme treats the entire `std::qsort` function as a single entity and expects it to have the same spectrum throughout its execution. We expect that this problem can be overcome by also adding loop markers to library code. Overall, the arithmetic mean for miss-recognition is 2.26%.

To provide further evidence that the ability to do Spectral Profiling is a result of a fundamental connection between repetitive program behavior and the spectra of resulting side-channel signals, we apply Spectral Profiling power signals produced via cycle-accurate architectural simulation in SESC [92], a cycle accurate simulator that includes CACTI [93] and WATTCH [94] power models. In this simulation, we model a 1.8GHz 4-issue out-of-

order core with 32KB L1 and 64MB L2 caches. Using this simulator, we found similar results (98% accuracy on average) to that of in real measurements. This is slightly better than our real-system results, mainly because simulation-produced power signals are free of radio-frequency noise and other problems that present in EM signals received from real systems are: measurement error, frequency-dependent distortion (for some EM frequencies the real system acts as a better "transmitter" than for others), etc. The mean for miss-recognition in simulations is 1.19% which is again slightly better than for the real system.

Overall, Spectral Profiling remains effective in spite of differences in clock rates (1.008 GHz vs. 1.8 GHz), pipelines (In-order vs. Out-of-order), use of different signals (EM emanations vs. power), etc. This provides strong evidence that the existence of spectral signatures is not an anomaly of the particular system we used in our experiment, and that Spectral Profiling is likely to be effective for a wide variety of other real computer systems.

3.6 Conclusions

In this chapter we demonstrated how Spectral Profiling method can be used to develop a framework which profile program execution without instrumenting or otherwise affecting the profiled system. Spectral Profiling monitors EM emanations unintentionally produced by the profiled system, looking for spectral "spikes" produced by periodic program activity (e.g., loops). This allows Spectral Profiling to determine which parts of the program have executed at what time and, by analyzing the frequency and shape of the spectral "spike", obtain additional information such as the per-iteration execution time of a loop. The key advantage of Spectral Profiling is that it can monitor a system as-is, without program instrumentation, system activity, etc. associated with the profiling itself, i.e., it completely eliminates the "Observer's Effect" and allows profiling of programs whose execution is performance-dependent and/or programs that run on even the simplest embedded systems that have no resources or support for profiling. We evaluated the effectiveness of Spectral Profiling by applying it to several benchmarks from MiBench suite on a real system. Our

experimental results showed that our current implementation of Spectral Profiling on average correctly attributes 93% of execution time when applied to EM emanations from an actual IoT device, and we confirmed the versatility of the approach by also successfully applying it to the power signal produced through cycle-accurate simulation.

Overall, Spectral Profiling can be used for profiling in systems where infrastructure is not available, or where profiling overheads may perturb the results too much ("Observer's Effect").

CHAPTER 4
EDDIE: EM-BASED DETECTION OF DEVIATIONS IN PROGRAM EXECUTION

4.1 Abstract

Building upon leveraging AM-modulated EM side-channel signals for useful purposes, in this chapter we present our novel method called *EDDIE*, that leverages EM side-channels for intrusion detection. Similar to *Spectral Profiling* framework, the main advantage of this method is that it incurs zero-overhead on the system and can monitor the system *externally* which makes it suitable for resource-constrained (e.g., IoTs, CPSs, etc.) or legacy systems (which lack monitoring infrastructures). In addition, it provides a security air-gap, i.e., it is physically separated from the monitored device thus, for a successful attack, the attacker needs to attack both the monitored device and the intrusion detection system, and consequently makes the entire system more secure.

Monitoring with *EDDIE* involves receiving EM emanations that are emitted as a side effect of execution on the monitored system, and it relies on spikes in the EM spectrum that are produced as a result of periodic (e.g., loop) activity in the monitored execution (i.e., AM-modulated signals as it was described in Chapter 2). During training, *EDDIE* characterizes normal execution behavior in terms of peaks in the EM spectrum that are observed at various points in the program execution, but it does not need any characterization of the malware or other code that might later be injected. During monitoring, *EDDIE* identifies peaks in the observed EM spectrum, and compares these peaks to those learned during training. We evaluate *EDDIE* on a real IoT system and in a cycle-accurate simulator, and find that even relatively brief injected bursts of activity (a few milliseconds) are detected by *EDDIE* with high accuracy, and that it also accurately detects when even a few

instructions are injected into an existing loop within the application.

We believe that *EDDIE* would be particularly useful as a dedicated monitor for systems embedded in critical infrastructure, such as industrial, power plant, and other control systems, or for auditing the behavior in-body medical devices when a patient visits the doctor's office. In both scenarios, software changes and direct manipulation of devices is highly undesirable, while the cost of an *EDDIE* setup is very low compared to the total cost of the monitored system(s).

The main contributions of this work are:

- Proposing *EDDIE*, a new approach to monitoring execution and detecting anomalies without any modification to or cooperation from the monitored system.
- A proof-of-concept implementation of *EDDIE* that demonstrates its potentials.
- A detailed characterization of *EDDIE* implementation in the context of code injection, showing that *EDDIE* can detect injected code even if a brief (a few milliseconds) burst of injected code is executed, and that it can detect injections of just a few instructions within a loop body.

4.2 Design Overview

The overall idea of *EDDIE* is to use the observed EM spectra over time as a surrogate for program behavior over time, gather training data about what the EM spectra should look like in each part of the program during correct execution, and then monitor execution by looking for situations where the observed EM spectra statistically deviate from expected spectra, i.e., the observed spectra are unlikely to be outcome of a correct execution.

EDDIE obtains the EM signal from an antenna, uses the Short-Term Fourier Transform (STFT) to convert this continuous signal into a sequence of overlapping windows, and then converts the signal in each window into its spectrum, which we call Short-Term Spectrum (STS). All the actual training and monitoring in *EDDIE* is done on this sequence of Short-

Term Spectra (STSs). Training in *EDDIE* consists of obtaining a number of STSs for each loop nest and each loop-to-loop transition that is possible during a valid execution in the program. During monitoring, *EDDIE* compares the observed STSs to those obtained during training, and reports a problem when the observed sequence of STSs is unlikely to have been produced by a valid execution.

Compared to directly using the corresponding time-domain signal values, use of STSs in *EDDIE* is advantageous both in terms of efficiency and in terms of accuracy. During monitoring, *EDDIE* needs to assess whether the difference between the monitoring-time signal and the training-time signal is larger than the difference that can be expected due to "usual" variation in the signal such as signal noise and measurement error, cache misses and other low-level events, etc. At any given time, the program is very likely to be executing a loop nest, so an STS is very likely to have a few prominent features (peaks) that are much stronger than the noise at that frequency, and whose position in the spectrum (frequency) is very resilient to "random" occurrences of low-level hardware events and completely unaffected by signal noise. This means that comparisons among STSs are usually very efficient because they involve checking only a few points (frequencies) in the spectrum, and it also means that STSs for the same region of code tend to be very similar to each other, so even small differences in STS peaks provide high confidence that something else is executing. In contrast, the time-domain signal for the same time window, typically consists of fluctuations that are relatively weak (compared to signal's noise level) and whose position in the time window shifts significantly (compared to the duration of each fluctuation) as a result of low-level hardware events. This means that comparisons would have to include most of the real-time samples collected in the time window, and that accuracy would suffer because relatively large differences among signals would have to be tolerated to avoid producing false positives.

At high-level, *EDDIE* has two phases: training, and monitoring. Followings describe these two phases in more detail. We use an application, *Susan*, from a standard benchmark

suite, MiBench [87], as a running example.

4.3 Training Phase

The main goal of the training phase is to (*i*) find the possible sequences in which loop and inter-loop regions may execute, and (*ii*) collect enough sample windows that correspond to each loop and inter-loop region of the program, along with information about which region each of the training window corresponds to.

The possible sequences of regions are represented as a loop-level state machine that is identified through compile-time analysis. This analysis begins with the traditional control flow graph (CFG) of the program. Each node in the CFG is a basic block, and an edge from some basic block, `A`, to some basic block, `B`, exists if execution of block `A` can immediately be followed by execution of block `B`. Note that the CFG defines a state machine that constrains the set of basic block sequences that may be observed in an execution of that program. To obtain the region-level state machine, for each loop nest we merge all the nodes in the CFG that belong to that loop nest into a single loop-region node, eliminating all edges between basic blocks inside that nest and also all edges that go from that nest directly to itself. We then eliminate each of the remaining basic-block nodes from the graph by connecting the sources of its incoming edges directly to the destinations of its successors, and finally we merge (into a single edge) those edges that have both the same source node and the same destination node. The result of this is a graph that represents the region-level state machine of the program: each state (graph node) represents a loop region and each edge represents an inter-loop region. Note that this region-level state machine is very compact compared to the traditional basic-block-level CFG, and that at runtime the state transitions in the region-level state machine occur much less often because each state represents execution of an entire loop nest.

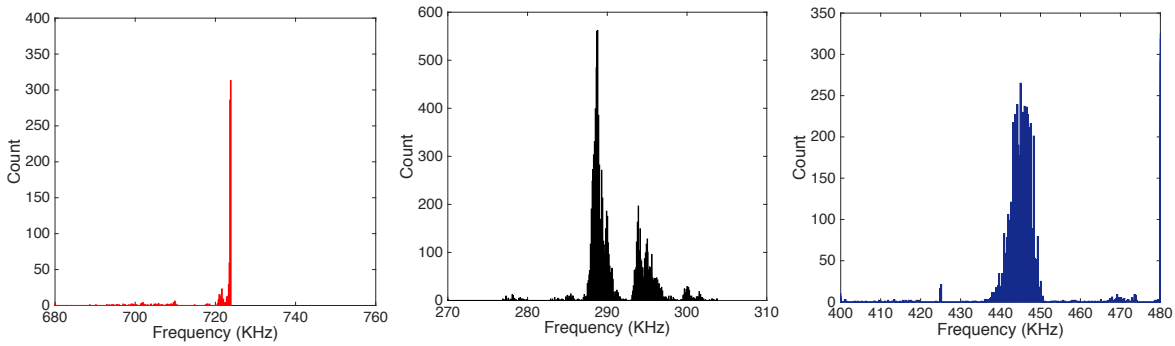
During compilation for training runs, *EDDIE* adds instrumentation just before and just after each loop nest in the code that will be used in training runs. In each *EDDIE* training

run, we record the signal while the instrumentation logs the region identifier, entry time, and exit time for each loop region that is executed. This allows us to map each part of the signal to the region that was executing at that time. This instrumentation is light-weight and requires very little memory to record its information. For example, only five loop nests are instrumented in the *Susan* application from MiBench benchmark suite [87].

EDDIE then runs the application multiple times, each time with different inputs. Multiple runs are needed to improve coverage of the regions, *i.e.*, to obtain signals that correspond to regions that are only executed in some of the runs. Multiple runs also help gather a representative number of signal windows for regions that exhibit variation in behavior, *e.g.*, due to control flow within the body of a loop. For example, our training for the *Susan* benchmark consists of 50 runs (each with different inputs).

The signals collected during training are then divided into sample windows. For each window *EDDIE* uses STFT to compute its spectrum, and then identifies the set of peak frequencies in that spectrum. A peak frequency in *EDDIE* is defined as a frequency at which at least 1% of the entire window's signal energy is concentrated. The number of peak frequencies can differ from window to window, especially if they correspond to different regions of the program. For example, for one loop nest in the *Susan* benchmark our training produces 1,200 windows, each with 15 peaks, while for another loop nest we have 750 windows, each with 7 peaks. Generally, we observe ***three types*** of STSs in the loops. *(i)* loops that have very small variations per-iterations and thus they have very sharp peaks. *(ii)* loops that has larger per-iteration execution time variations hence wider peaks due to these variations (usually these variations caused by different control-flow paths in the code). *(iii)* nested loops where inner loop can vary a lot thus the peak(s) for outer loop can be much wider than the other two types. These three types are shown in Figure 4.1. Using this observation, for each STS we take several peaks not just the strongest one.

Finally, for each region in the program, *EDDIE* forms the set of sample windows that belong to that region and performs analysis to determine how many samples need to be



(a) (b) (c)

Figure 4.1: Histogram of frequencies that correspond to per-iteration execution time ($f = 1/T$) for three different loops: (a) fixed. (b) control flow loop. (c) nested loop.

jointly considered in statistical tests during the monitoring phase to achieve a desired level of reporting accuracy. This number depends on the statistical test that will be used and is explained in the following section.

In summary, *EDDIE*'s training obtains the application's region-level state machine and, for each region, a reference set of sample windows with their spikes already identified, and the knowledge of how many samples should be used in statistical tests for that region.

4.4 Monitoring Phase

4.4.1 Distance Metric

A key aspect of *EDDIE*'s decision-making is that it cannot be based on *whether* the observed (monitoring-time) STSs differ from the reference (training time) STSs. This is because STSs that belong to the same region of code are almost never *exactly* the same. Reasons for this include noise and external (e.g., radio) interference in the signal, and also "random" variation in program behavior (e.g., a specific path through the loop body may be taken slightly more or less often depending on the window of time we are observing), micro-architectural events such as cache misses, branch mis-predictions, etc.

Due to variations among STSs that belong to the same region of code, *EDDIE*'s decision-making is based on **statistical tests**. This means that *EDDIE* views each code region as

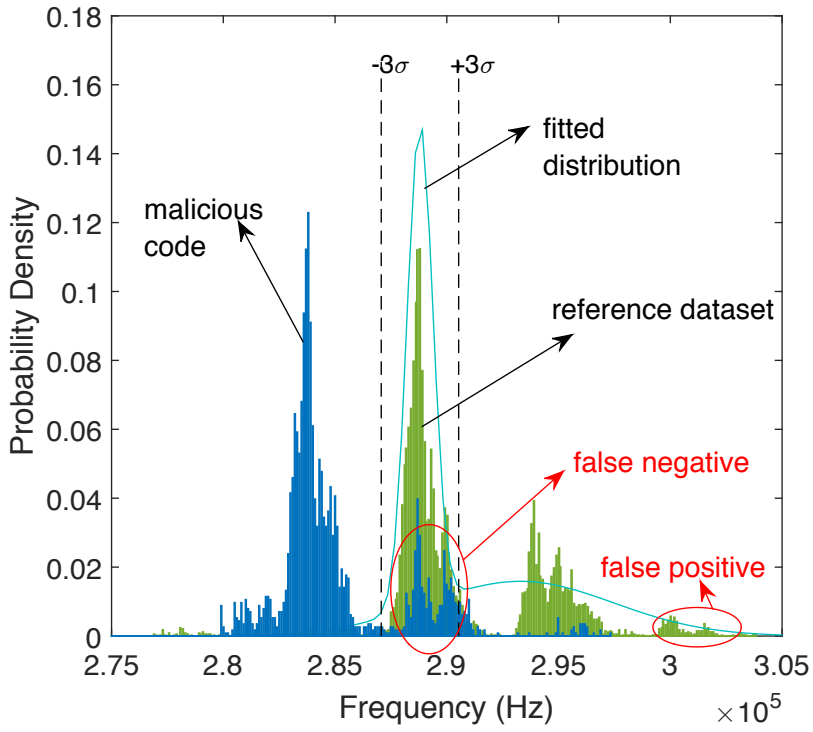


Figure 4.2: Normal (green) vs. Malicious (blue) activity. Parametric test can lead to inevitable false positives and false negatives.

a generator of STSs that vary randomly according to some distribution that is specific to that region. During monitoring, *EDDIE* uses an appropriate statistical test to compute the probability that the region's reference distribution (the distribution of STSs obtained for that region during training) is the same distribution that has produced the STSs observed during monitoring. If the same-distribution probability is high enough, *EDDIE* considers the observed STSs to be "as expected" and takes no further action. Conversely, if the same-distribution probability is low enough, *EDDIE* considers the observed samples as anomalous and takes further action.

The simplest statistical tests are called *parametric* tests because they assume that the distribution belongs to a specific family, e.g., Gaussian (normal) distributions, and can be fully characterized using a relatively small set of parameters (e.g., mean and standard deviation for a normal distribution). However, we have found that many regions of code produce distributions that are poor matches for well-known distribution families. For ex-

ample, Figure 4.2 uses green color to show how the frequency of the strongest spike is distributed among reference STSs that all belong to the same loop nest. This distribution is a poor fit for a normal distribution, so for illustration purposes we show (light blue line) the best-fitting *multi-modal* distribution that consists of two normal distributions. This bi-normal fitted distribution differs significantly from the actual (green) distribution, so statistical tests that assume a bi-normal distribution would report many false positives because the observed distribution of STSs *does* differ from the reference bi-normal distribution.

To overcome this problem, *EDDIE* uses a *nonparametric* test to compare the observed and reference STS distributions. A nonparametric test can take two groups of data and compute the probability that the two groups are both random samplings from the same population, without any a-priori assumptions about the nature of that population's underlying distribution. For *EDDIE*, this means that we can test the observed STSs against reference STSs without making any assumptions about which type of distribution the reference STSs belong to. Two best-known non-parametric tests are the *Wilcoxon-Mann-Whitney test* (*U-test*) [95] and the *Kolmogorov-Smirnov test* (K-S test) [96]. The U-test is sensitive to the differences in the median value in each of the two groups of data and the K-S test is sensitive to any difference between the two groups of distribution. We experimented with both tests and found that the **K-S test** shows better performance, so *EDDIE* uses that test.

In K-S test, we suppose that the reference data set has m elements with an empirical distribution function of $R(x)$, and that the data set observed during monitoring has n elements with an empirical distribution function of $M(x)$. The K-S test then computes $\mathrm{D}_{m,n} = \max_x \mid R(x) - M(x) \mid$, i.e., the largest difference between these two empirical distributions. The null hypothesis, H_0, is *the reference (training-time data) and monitoring-observed data sets were both drawn from the same population*. The test rejects this H_0 at significance level α if $\mathrm{D}_{m,n} > \mathrm{D}_{m,n,\alpha}$, where $\mathrm{D}_{m,n,\alpha} = c(\alpha)\sqrt{\frac{m+n}{mn}}$ for sufficiently large m and n, and where $c(\alpha)$ is the reverse of Kolmogorov-Smirnov distribution at confidence level α. Intuitively, $\mathrm{D}_{m,n,\alpha}$ is the magnitude of the difference that can be expected to exist

between $R(x)$ and $M(x)$ even when the two data sets *are* drawn from the same population, and α is the fraction of test in which the test is allowed to falsely reject H_0. Note that α cannot be zero because for any value of $\mathrm{D}_{m,n}$ there exists a small possibility that it can occur even when H_0 is true.

A final consideration for K-S test is that K-S test the is a one-dimensional test, i.e., the data sets it can compare should have elements that are scalars (numbers), while in *EDDIE* a STSs are characterized by a set of peak frequencies (a vector of numbers). Thus we apply the K-S test to each dimension separately, i.e., one test compares the STSs according to their strongest peak, a second K-S test compares the STSs according to their second-strongest peak, etc. The results of these tests are then combined by counting the number of such tests that have rejected H_0. Followings discuss this in more detail.

4.4.2 Non-parametric Statistical Test

A final consideration we need in order to use the K-S test in *EDDIE* is that the K-S test is a one-dimensional test, i.e., the data sets it can compare should have elements that are scalars (numbers), while in *EDDIE* a STSs are characterized by a set of peak frequencies (a vector of numbers). Thus we apply the K-S test to each dimension separately, i.e., one test compares the STSs according to their strongest peak, a second K-S test compares the STSs according to their second-strongest peak, etc. The results of these tests are then combined by counting the number of such tests that have rejected H_0.

In the K-S test, we can choose n, the number of monitoring-observed STSs that will be tested in each K-S test. With a small n, the K-S test uses only very recently observed STSs so *EDDIE* will have a low latency between when the anomalous execution begins and when it is detected. This *detection latency* can generally be expected to grow in proportion to the value of n. However, n also affects detection accuracy, and the relationship between accuracy and the value of n is not as straightforward. To illustrate this, Figure 4.3 shows the false rejection rate in *EDDIE*'s K-S test, i.e., how often the K-S test fails in an injection-free

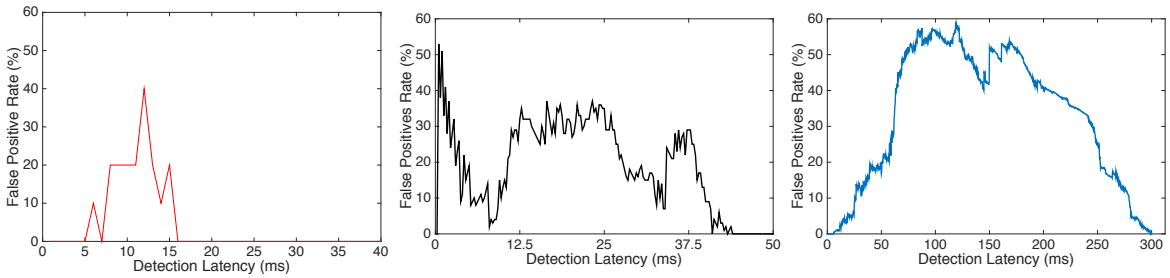


Figure 4.3: Buffer size selection for three loops, one whose spectrum has one sharp peak and its harmonics (left), one whose spectrum has several peaks and their harmonics (middle), and one whose spectrum has poorly defined peaks (right).

run, as we increase n, which is shown in terms of detection latency (in milliseconds).

When n is too small, for most regions the K-S test will rarely reject H_0 even when H_0 is actually false. This is not surprising because, intuitively, the test does not have enough monitoring-observed data points to reach a sufficiently high confidence level. As n increases, the rejection rate usually increases for both correct rejection (the STSs come from the injected part of the execution, not shown in Figure 4.3) and false rejection. At some point, the correct rejection rate becomes very high (most STS sets that include injections are rejected by the K-S test) but the false rejection rate is also relatively high. Further increase of n then results in virtually no change in correct rejections, but the false rejection rate drops and eventually becomes very close to zero.

We want *EDDIE* to be useful in practice, so its reports should have very few false positives. Although *EDDIE*'s overall algorithm has additional considerations beyond the K-S test rejecting a group of STSs, it would be very hard to achieve a very low false positive rate if the K-S test labels a significant percentage of monitoring-time STS groups as anomalous. Thus in *EDDIE* we should use the K-S test with the number of monitoring-observed STSs n that is large enough to provide a near-zero false rejection rate, but not much larger than that because that unnecessarily sacrifices detection latency. Unfortunately, as shown in Figure 4.3, this "sweet spot" value of n depends on the code region and differs quite a bit from region to region.

To accommodate this, during training *EDDIE* determines the desired value of n separately for each region. We select the desired n by applying the K-S test to training-time STSs for that region using different values of n, and selecting the smallest n that provides the minimum false rejection rate observed across the entire range of ns (for most regions the false rejection does reach zero at some value of n).

Note that all training runs are injection-free, so this method can only consider the false rejection rate, but we have found that this also results in near-optimum correct rejections.

4.4.3 Monitoring Algorithm

Algorithm 1 shows (with some simplifications) how *EDDIE* works and switches between regions and decides when to report an anomaly to the user.

The loop at Line 7 iterates over STSs observed during monitoring. The new STS is added to the set that will be used in the K-S test, and the oldest STS in the set is removed to maintain the set at the appropriate size for the current region. The K-S test is then used, one peak at a time, to compare the monitoring-observed set ($MonSet$, see Line 9) to the reference set for the current region. When the K-S test rejects the null hypothesis, *EDDIE* considers the possibility that the execution has progressed beyond the current region. This is also done using the K-S test, by comparing the appropriate number of monitoring-observed STSs to the reference STSs for each candidate region. Acceptances and rejections across all peaks are counted. After all candidates have been considered, if a next-region candidate had enough peaks accepted by the test, *EDDIE*'s current region is changed to that candidate region. If multiple candidates have enough peaks accepted (this happens extremely rarely), *EDDIE* uses the candidate region with the most accepted peaks. If none of the next-region candidates is acceptable, *EDDIE* checks if the number of recent rejections in the K-S test is high enough (i.e., 3 in our implementation) to report an anomaly to the user. This allows *EDDIE* to tolerate a few K-S test rejections without reporting anything, which helps reduce the number of false reported anomalies that can occur in injection-

Algorithm 1 *EDDIE*'s intrusion detection algorithm

Regions: $R_1 ... R_r$
Current region number: c
Group size for K-S test: $n_1 ... n_r$
$P(i, j)$: jth peak in the the ith STS
$pos \leftarrow 1$; $counter \leftarrow 0$;
$currRegion = R_1$
while application is running **do**
    for p **do**=1..numPeaks(R_c)
        $MonSet \leftarrow P(pos - n_c : pos, p)$
        if test($RefSet_{c,k}, MonSet) = reject$ **then**
            for $R_j \in$ successors of R_c **do**
                $AltSet \leftarrow P(pos - n_j : pos, p)$
                if test($RefSet_{j,k}, AltSet) = reject$ **then**
                    $anomalyCnt \leftarrow anomalyCnt + 1$
                else
                    $changeCnt(j) \leftarrow changeCnt(j) + 1$
                end if
            end for
        elseReset $anomalyCnt$ and $changeCnt$
        end if
    end for
    if $changeCnt > changeThreshold$ **then**
        $j \leftarrow$ index of $max(changeCnt)$
        $currRegion \leftarrow j$
    end if
    if $anomalycnt > reportThreshold$ **then**
        Report anomaly to user
    end if
    $pos \leftarrow pos + 1$
end while

free runs when, for example, interrupts and other system activity occasionally result in a "deviant" STS.

4.5 Experimental Results

4.5.1 Experimental Setup

We use two different experimental setups. One is a real IoT prototype system, a single-board Linux computer (A13-OLinuXino-MICRO [97]) with a 2-issue in-order ARM Cor-

tex A8 processor with a 32kB L1 and a 256kB L2 cache, with a Debian Linux operating system. The EM signals emanated from this system are received by a commercial small electric antenna (PBS-E1 [89]) that is placed right above the device's processor, and the signal is recorded using a Keysight DSOS804A oscilloscope. While this oscilloscope is relatively expensive, note that we use it mainly because of its built-in features for automated and calibrated measurements and ability for displaying the real-time signals. In additional experiments, we have observed similar EM spectra with less expensive ($<$\$800) commercial software-defined radio receiver (USRP b200-mini) and we confirm that *EDDIE* can work efficiently on such lower-cost setups. While this cost is low enough for deploying *EDDIE* in some important scenarios (critical infrastructure, medical offices, etc.), for other scenarios we envision a custom design with a specialized receiver (ASIC block for STFT and peak finding, simple CPU for tests, and some flash for storing the model from training) attached to an antenna, with a $<$\$100 total cost.

Our second setup is based on the SESC [92] cycle-accurate simulator, and uses the simulator-generated power signal for *EDDIE*'s analysis. This setup is used to confirm that *EDDIE* is applicable across a wide range of systems, and to gain insight into which architectural features affect *EDDIE*'s detection performance.

In our experiments, we use a total of 10 benchmarks from the MiBench [87] suite to test *EDDIE* algorithm. For the real IoT system, we execute each benchmark 25 times during training. The code for the training runs contains with our light-weight instrumentation, which is implemented as a Clang tool, and the code is also subjected to a separate analysis (which is not used to actually generate code) in LLVM [98] where we added a pass that statically finds the regions and the possible transitions between regions. For monitoring, we use 25 runs per benchmark, without any instrumentation and with different inputs.

In simulation-based experiments we use fewer runs (10 training and 10 monitoring runs per benchmark) to reduce the overall simulation time.

Table 4.1: Accuracy for *EDDIE* monitoring of an actual IoT device

Benchmark	Detection Latency (ms)	False Positives (%)	Accuracy (%)	Coverage (%)
Bitcount	42	0.99	100	99.9
Basicmath	25	1.8	99.9	99.9
Susan	32	1.39	92.1	95.9
Dijkstra	25	1.08	99.9	99.7
Patricia	28	0.98	92.3	95.2
GSM	24	0.9	96.2	57.1
FFT	17	0.76	93	99
Sha	11	1.9	97.2	98.9
Rijndael	12	0.56	99.9	97.1
Stringsearch	11	0.19	99.9	99.9

4.5.2 Results on an IoT Device

In this set of experiments, we inject code into different regions of each application. The injections are different for loop and inter-loop regions. Injections outside loops consists of invoking a shell and then, without doing anything else, returning back to the original application. This injection results in executing 476k injected instructions and adding about 3 ms to the execution time. When injection is made in a loop, we add an 8-instruction code that consists of 4 integer operations and 4 memory accesses. The rationale for the shellcode injection is that shellcode execution is often a fundamental step in many attacks, and our empty-shellcode injection results in less injected-code execution than any real shellcode-based attack where the attack's intended activity (payload) must either be executed or at least set up within the shellcode-invoked shell. The rationale for injecting only 8 instructions into a loop body is that an injection into a loop allows the injected code to be executed repeatedly, allowing the attacker to perform significant work over time but improve stealth by performing the work in small chunks.

The results for the IoT system are shown in Table 4.1. The first column shows the application, and the remaining columns report *EDDIE*'s detection latency, false positive and accuracy percentages, and coverage. The results were obtained using `reportThreshold`

= 3 in *EDDIE*'s algorithm, i.e., *EDDIE* tolerates up to 3 consecutive K-S test rejections and only reports an anomaly for a rejection that is part of a 4-long (or longer) streak of test rejections. The average detection latency is measured as the average, among all injections that are reported, of the difference between when execution of injected code begins and the time when *EDDIE* reports it. This latency mainly reflects the number of STSs that are used in the K-S test (n in Section 4.4.1). False positives are the number of STS groups that are reported as anomalous but do not contain any injected execution, as a percentage of all STS groups. The average for false positives is $<1\%$ and the highest false positive percentage was only 1.9% (for the *Sha* benchmark). Accuracy is computed for each region as the total number of STS groups with a correct reporting outcome, i.e., those that contain injections and are reported by *EDDIE* plus those that contain no injections and are not reported, expressed as a percentage of all STS groups. The accuracy shown for each benchmark is the average of its per-region accuracy results. On average, *EDDIE*'s accuracy is 95%. We observed that the bulk of the inaccuracies come from borders between two regions (i.e., outside the loops), and further investigation has revealed two main causes for this: (i) non-loop code during some transitions creates poorly defined peaks, so better consideration of diffuse spectral features may improve *EDDIE*'s accuracy, and (ii) the actual inter-loop transition is usually very brief and for different executions occurs at a different point in the window on which the STS was computed, so better identification of the boundaries of the actual inter-loop transition may help to create STSs that better represent the transition. Finally, we define coverage as the amount of time during which the STS is attributed to the region in the code that actually produced it. The main reason for imperfect coverage in our implementation is that some loops have no peaks in their STSs. For example, about 40% of the execution time in *GSM* is spent in one such loop, and this accounts for nearly all of its poor coverage.

4.5.3 Simulation Results and Sensitivity to Processor Architecture

To gain more confidence that *EDDIE* is a broadly applicable approach, and to get more insight into which aspects of the system's architecture have an effect on *EDDIE*'s accuracy, we apply *EDDIE* to the power consumption signal generated by the SESC simulator with integrated CACTI [93] and WATTCH [94] power models for its cache and configurable pipeline. We first model a 1.8 GHz 4-issue out-of-order core with 32KB L1 and 64MB L2 caches, the power signal provided to *EDDIE* is sampled every 20 cycles, and *EDDIE*'s STFT uses 0.1ms windows with 50% overlap. The code injection in these simulations is implemented by directly injecting dynamic instructions into the simulated instruction stream without changing the application's code or using any architectural registers. This maximizes the injection's stealth and is an idealized representative of an attack that uses only registers that are dead at the injection point in the original application.

Table 4.2: *EDDIE*'s latency and accuracy when using a simulator-generated power signal

Benchmark	Average Latency	False Rejection	Accuracy	Coverage
Bitcount	7ms	0.8%	99.9%	99.9%
Basicmath	8ms	0.2%	99.9%	100%
Susan	5ms	0.7%	91.4%	96.6%
Dijkstra	10ms	0.3%	97.02%	99.9%
Patricia	13ms	0.4%	94.14%	98%
GSM	6ms	0%	100%	68.3%
FFT	5ms	0.4%	97.8%	99.1%
Sha	0.4ms	1.83%	100%	100%
Rijndael	0.6ms	0.24%	97.1%	97.2%
Stringsearch	0.2ms	0%	100%	100%

EDDIE's results for these simulation-based experiments are shown in Table 4.2. False rejections occur on average in 0.7% STSs, an expected improvement over real-system experiments because the simulation has no signal noise, no interrupts or other system activity, etc. By comparing results from simulation and real-system experiments, we can also conclude that *EDDIE*'s accuracy and detection latency are more affected by the applications

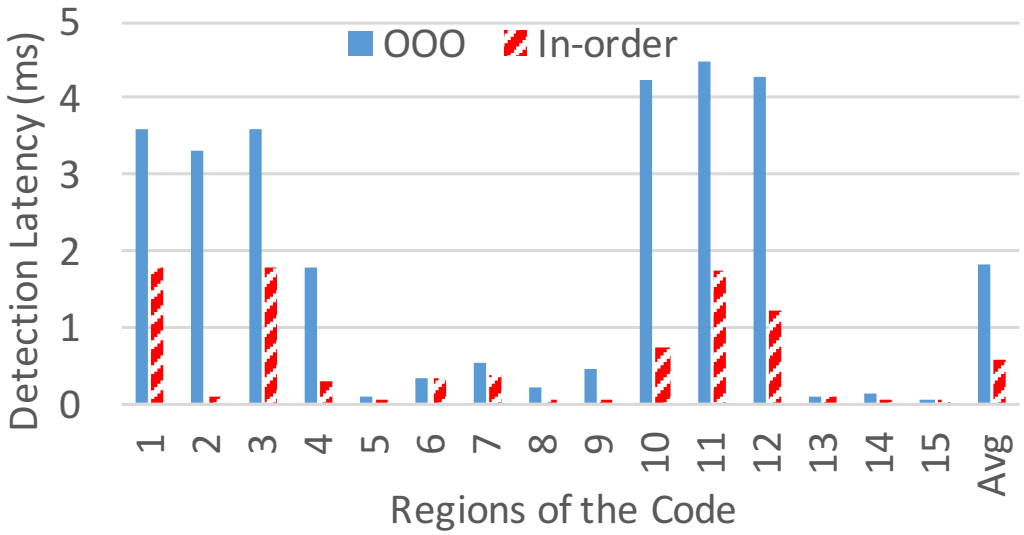


Figure 4.4: Detection latency of 15 different regions in in-order and out-of-order architecture.

itself (i.e., mostly the shape of the spectrum for the code regions) than by factors such as noise, interference, etc.

Intuitively, we expect *EDDIE* to perform better on systems whose architecture introduces less variation in executions of the same region of code. To get more insight into which architectural parameters have a significant impact on *EDDIE*'s detection performance, we configure the simulator to model an in-order processor with 3 different issue widths (1,2, and 4) and 2 different pipeline depths, and an out-of-order processor with 3 issue widths (1,2,4), 3 pipeline depths, and 5 ROB sizes, for a total of 51 configurations. We then simulate execution of 3 benchmarks (*Basicmath*, *Bitcounts*, and *Susan*) on each configuration and use N-way analysis of variance (ANOVA) to determine which factors have a significant impact on *EDDIE*'s results.

We found that for out-of-order and in-order architectures *EDDIE* achieves similar false rejection and accuracy results, but its latency (i.e., number of STSs that need to be considered in the K-S test) is significantly higher for out-of-order architectures (see Figure 4.4) because an out-of-order core tends to produce more variation in its dynamically constructed instruction schedule, creating more variation among STSs and thus requiring more STSs to

capture their distribution.

We also found that in in-order architectures pipeline depth and issue width have no statistically significant effect on *EDDIE*'s results, and that in out-of-order architectures the ROB size and issue width also have no statistically significant impact on *EDDIE*'s results. However, in out-of-order processors pipeline depth has a weak but statistically significant impact on detection latency. A closer look at the data reveals that in 27% of the code regions pipeline length increases detection delay, and that these affected regions are all loops with control-flow variation among iterations, so the likely explanation for increased detection delay in *EDDIE* is that a deeper pipeline results in more timing variation due to branch mis-predictions, that in turn increases the size of the STS group that representatively captures this variation (and the n for the K-S test).

Finally, we repeated this analysis for different amounts of injected execution, and found that the impact of pipeline depth in out-of-order processors on *EDDIE*'s results diminishes as the injection size increases, and for large-enough injections the pipeline depth no longer has a statistically significant impact of *EDDIE*'s detection latency. This means that large amounts of injected activity can be detected quickly even when the processor's pipeline is deep, but for smaller injections longer pipelines result in longer detection latency.

4.5.4 Effect of the Execution Rate of Injected Code

An intuitive way to improve stealth is to further diffuse injected execution by injecting the code inside a loop body such that only some loop iterations execute (a small amount of) the injected code. To evaluate *EDDIE* in this context, we use our simulator-based setup and for the targeted loop region randomly choose the iterations that will be injected with 8 memory instructions and 8 integer operations. We use *contamination rate* to refer to the percentage of iterations that contain injected execution, and we repeat this set of experiments for contamination rates between 100% (where every iteration is injected) and 10% (where 90% of the iterations are injection-free).

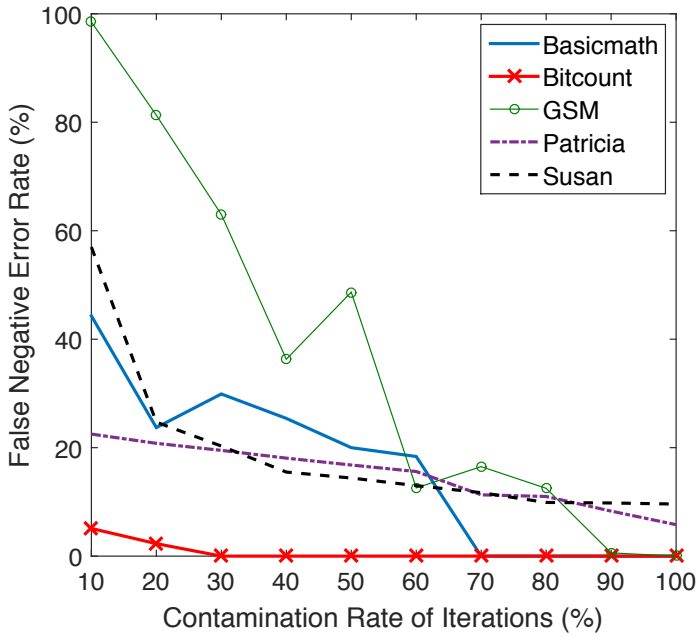


Figure 4.5: False negative rate of variable injection rates.

Figure 4.5 shows the *false negatives*, i.e., the percentage of injection-containing STSs that are not reported by *EDDIE*, for different contamination rates. As expected, *EDDIE*'s ability to detect the injection does diminish with the injection's contamination rate, but for most applications *EDDIE* still retains significant ability to detect injections even at low contamination rates. For example, for *Bitcount*, *EDDIE* still detects >90% of injection-containing STSs even when only 10% of loop iterations actually contain injected execution. However, in *GSM*, *EDDIE* detects only 5% of the STS at the 10% contamination rate. Note that this does not mean that *EDDIE* is inherently unable to detect injections that have low contamination rates. Indeed, Figure 4.6 shows the results in terms of detection latency (which is increased by increasing n in *EDDIE*'s K-S test) that is needed to maintain *EDDIE*'s accuracy. This indicates that *EDDIE* can very accurately detect even low-contamination-rate injections, but that detection of low-contamination-rate injections will have a longer latency.

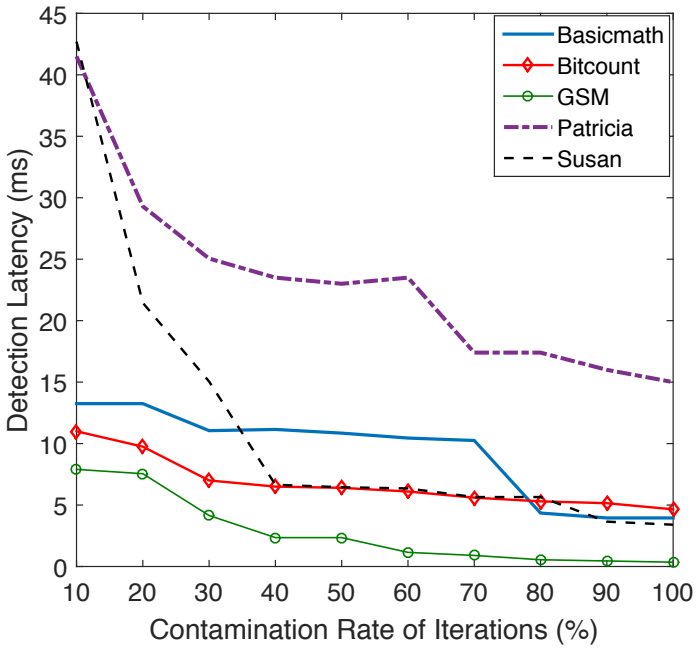
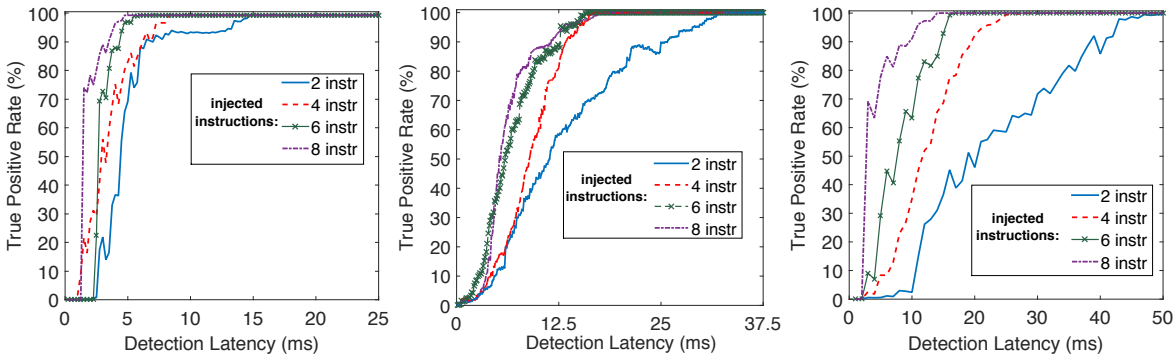


Figure 4.6: Detection latency of variable injection rates.

4.5.5 Size of Injection

In this section, we analyze the impact of the number of injected instructions on *EDDIE*'s detection accuracy. Our analysis considers injection *inside* and *outside* of loops separately. This is because even a few instructions injected inside a loop can accomplish significant work for the attacker as the injected code is executed many times during the loop. Outside loops, however, a successful attack usually requires injection of many instructions (recall that even an empty-payload shellcode executes over 500,000 instructions).

Figure 4.7 shows how *EDDIE*'s accuracy changes with the number of static instructions injected inside a loop. The smallest injection in this experiment consists of only two instructions: a store and an add. The remaining three injection sizes consists of 4, 6, and 8 instructions with the same instruction mix (equal number of stores and adds). To show how these results are affected by the loop's spectrum, Figure 4.7 shows the results for the same three loops used in Figure 4.3, i.e., a loop whose spectrum contains only one sharp peak and its harmonics, a loop with several less well defined peaks (and their harmonics), and a loop with a very diffuse peak that can be more accurately described as a hump. The figure



(a) (b) (c)

Figure 4.7: Accuracy when changing the number of injected instructions inside loops.

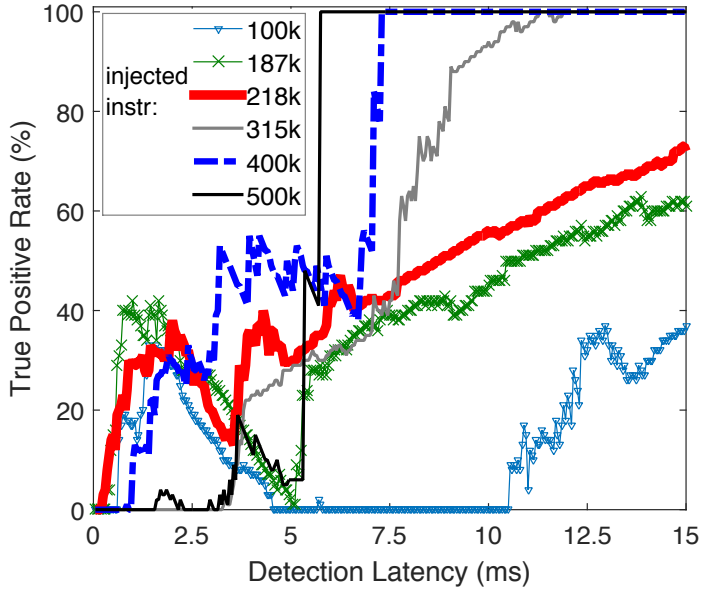


Figure 4.8: Accuracy when changing the number of injected instructions outside loops.

shows that even two-instruction injections into the loop body can be detected by *EDDIE* with extremely high accuracy, but that smaller injections have longer detection latency (i.e., use of a larger n in *EDDIE*'s K-S test).

Figure 4.8 shows the results for *outside* the loops. We use several different size of injections. In order to inject the code, we use an empty loop and put it between loop 2 and 3 in *Bitcount* application and change the number of iterations of this empty loop. As this number increased, number of injected instructions increased too.

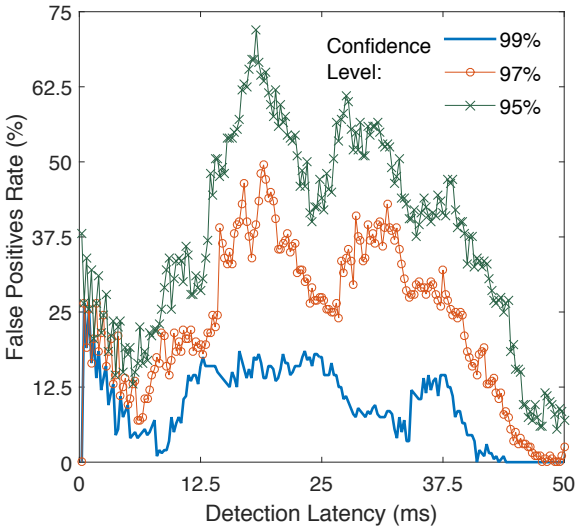


Figure 4.9: False positives in *EDDIE* for different K-S test confidence levels.

4.5.6 Effect of Changing the Confidence Level in the K-S Test

As mentioned earlier, we use KS-test to identify when the monitoring and the reference STSs differ too much. One important parameter in statistical tests is the confidence level, which introduces a fundamental trade-off between the test's false rejections (detecting an anomaly that does not really exist) and false acceptances (not detecting a real anomaly).

Figure 4.9 shows three confidence levels in terms of how the false positive rate changes with detection latency. As we can see, the 99% confidence level (which we use in all our experiments except this one) results in fewer false positives and it practically eliminates the false positives with a reasonable latency. Lower confidence levels result in many false positives at low latencies, and even at high latencies may not reduce false positives to acceptable levels.

4.5.7 Effect of Changing Instruction

In order to show how different types of injected instructions can affect detection latency and accuracy, we perform experiments in which we inject two different instructions in a loop. In the first set, we inject eight add instructions, while in the second set we inject 4

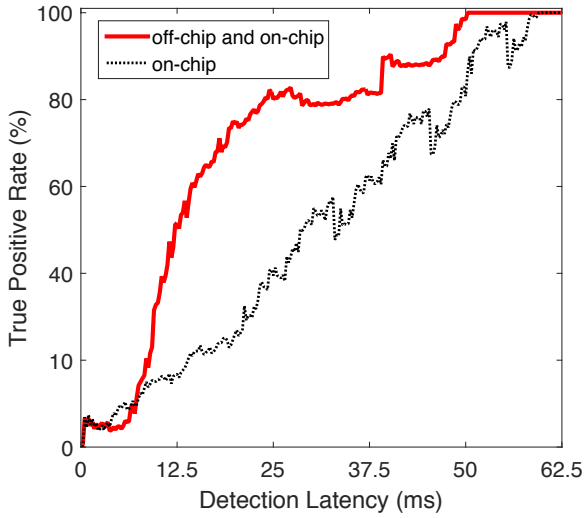


Figure 4.10: Effect of changing the type of injected instructions on latency and accuracy.

add instructions and 4 store instructions that randomly access a relatively large array so they often experience a cache misses.

Figure 4.10 shows the results for these two sets of experiments. These results indicate that instructions that result in off-chip activity tend to make the injection more visible and thus easier to detect quickly, but that even purely on-chip injections can be detected, albeit with an increased detection latency. In additional experiments we used other on-chip instructions like MUL, DIV, etc. but the results were very similar to those of the ADD instruction.

4.6 Conclusions

In this chapter we presented EM-Based Detection of Deviations in Program Execution (EDDIE), a new method for detecting anomalies in program execution, such as malware and other code injection, without introducing any overheads, adding any hardware support, changing any software, or using any resources on the monitored system itself. Monitoring with *EDDIE* involves receiving electromagnetic (EM) emanations that are emitted as a side effect of execution on the monitored system, and it relies on peaks in the EM spectrum that are produced as a result of periodic (e.g., loop) activity in the monitored execution. During

training, *EDDIE* characterizes normal execution behavior in terms of peaks in the EM spectrum that are observed at various points in the program execution, but it does not need any characterization of the virus or other code that might later be injected. During monitoring, *EDDIE* identifies peaks in the observed EM spectrum, and compares these peaks to those learned during training. Since *EDDIE* requires no resources on the monitored machine and no changes to the monitored software, it is especially well suited for security monitoring of embedded and IoT devices. We evaluated *EDDIE* on a real IoT system and in a cycle-accurate simulator, and found that even relatively brief injected bursts of activity (a few milliseconds) were detected by *EDDIE* with high accuracy, and that it also accurately detected anomalies when even a few instructions were injected into an existing loop within the application.

CHAPTER 5
REMOTE: ROBUST EXTERNAL MALWARE DETECTION FRAMEWORK BY USING ELECTROMAGNETIC SIGNALS

5.1 Abstract

As shown in the previous chapter, our presented framework, *EDDIE*, proposed to externally monitor IoT devices using electromagnetic emanations generated by the device. *EDDIE* appears to be very effective when trained on the same device it will monitor, under the same environmental conditions (ambient temperature, EM noise, etc.), with laboratory equipment, and with full availability of the source code for the software that will be monitored. However, to make external EM-based monitoring practical, it must support training on software binaries (no source code available for analysis), be robust to variations introduced by device manufacturing and environmental conditions to allow monitoring of various device instances and in environments different from those use for training, and be applicable to various IoT systems that have different hardware and system software.

To tackle these problems, this chapter proposes, REMOTE, a new robust framework to detect malware by externally observing electromagnetic (EM) signals emitted by Cyber-Physical System (CPS) while running a known application, in real-time and with a low detection latency, and without any a priori knowledge of the malware.

To demonstrate the usability of REMOTE in real-world scenarios, we port *two* real-world programs (an embedded medical device and an industrial PID controller), each with a meaningful attack (a code-reuse and a code-injection attack), to *four* different hardware platforms. We also port shellcode-based DDoS and Ransomware attacks to *five* different standard applications on an embedded system. To further demonstrate the applicability of REMOTE to commercial CPS, we use REMOTE to monitor a *Robotic Arm*. Our results on

all these different hardware platforms show that, for all attacks on each of the platforms, REMOTE successfully detects each instance of an attack and has $< 0.1\%$ false positives. We also systematically evaluate the robustness of REMOTE to interrupts and other system activity, to signal variation among different physical instances of the same device design, to changes over time, and to plastic enclosures and nearby electronic devices. This evaluation includes hundreds of measurements and shows that REMOTE achieves excellent accuracy ($< 0.1\%$ false positive and $>99.9\%$ true positive rates) under all these conditions. We also compare REMOTE to our prior work presented in the previous chapter, *EDDIE*, and demonstrate that this prior work is unable to achieve high accuracy under these variations and that is why REMOTE is designed.

The practical issues for monitoring of CPSs include:

- **Source Code and Infrastructure Availability**: application's source code, measurement, and/or instrumentation infrastructure might be **unavailable**,
- **Hardware and Software Variability**: the hardware might be based on different processor architectures and/or the device may use different operating systems or even not have one (bare-metal),
- **Physical Limitations**: enclosures may prevent direct access to the device, e.g., to place a power or EM probe very close to its processor and/or even system board,
- **Environmental Noise**: when using analog signals for monitoring, the environment may change the emanated signal and/or add interference to the received signal.

To develop REMOTE, we first understand how these issues may affect the EM signal, by carefully designing a signal measurement campaign on different systems, at different distances, under different environmental conditions. We then analyze these signals to understand in what ways they change and what they have in common. Using insights from this analysis, we design REMOTE to be robust to signal-related issues and to only rely on signal analysis (not the source code, instrumentation, etc.) for its training.

To emphasize practical applicability, our evaluation uses several real-world cyber-physical-systems: a medical embedded device called *Syringe-pump*, a controller for the temperature of a soldering iron, an IoT device while executing a set applications from a standard embedded benchmark suite, and finally, a commercial industrial robotic arm. We implement these applications on a variety of well-known CPS/embedded devices including an Arduino UNO board, Altera's FPGA Nios II soft-core, and two Linux mini-computers: OlimexA13 and TS-7250. For monitoring these applications, we also implement several meaningful attacks. For SyringePump, we implement a **Code Reuse Attack** by exploiting a buffer-overflow vulnerability that already existed in its open-source code. For the PID controller, we implement a *Stuxnet*-like **Advanced-Persistent Attack** where we assume the adversary has the ability to inject malicious instructions into the source code itself and modify the firmware of the system. Furthermore, we implement a **Shellcode Injection Attack** (i.e., gain shell access by overflowing a vulnerable buffer) on several applications from a standard embedded benchmark suite, MiBench [87], which represents usual activities of embedded /Internet-of-Things (IoT) systems in market. After hijacking the control-flow and invoking the shell, either a DDoS payload of the IoT botnet *Mirai* or encryption activity of a *Ransomware* payload is launched by the shell. We picked these two malwares since DDoS and Ransomware have become widespread threat and popular malware in recent years. We then show how REMOTE can find all the instances of these attacks with excellent accuracy. Lastly, for the Robotic-arm system, we use a commercial device (LewanSoul LeArm 6DOF [99]) and implement a **Firmware-modification/Zero-day** attack where we assumed that the libraries are compromised.

Specifically, the contributions of this work are:

- **Black-box training model and a new distance metric**: unlike existing approaches, our method can be trained and monitors the execution without requiring any access to the source code. Moreover, we propose a new distance metric that is robust against several kinds of variability.

- **Robustness**: our method is designed to be robust against a variety of hardware (e.g., ISA), software (e.g., OS), and environmental (e.g., temperature) variability. We will compare REMOTE to prior work, and show REMOTE can achieve much higher accuracy.
- **Real attacks and security analysis:** we present a variety of real attacks including a a code-reuse attack, an APT, a zero-day, and a shellcode injection attack.

We envision that REMOTE can be used in scenarios where the security of the CPS is critical, e.g., devices that control critical infrastructures, military systems, hospital equipment, etc. In these scenarios, the cost for deploying REMOTE is very low compared to the cost of the system it monitors, and the complexity of deploying REMOTE is relatively low because it requires no changes to the monitored device and thus creates no regulatory, safety, or disruption concerns for the system. For example, REMOTE is very suitable for scenarios like an industrial robotic arm or a CPS in a power-plant. An additional advantage of REMOTE over malware detectors that are implemented as part of the monitored system is that the REMOTE monitor is an entirely separate system, so it cannot be subverted by the same attack that succeeds in completely taking over the monitored system.

5.2 Design Overview

5.2.1 Spectral Samples (SS)

At a high level, REMOTE has two phases: training and monitoring. In both phases, the EM signal is first transformed into a sequence of spectral samples (SS) by using short-time Fourier transform (STFT), which divides the signal into equal-sized segments (*windows*), where consecutive segments overlap to some degree. STFT then applies the Fast Fourier Transform (FFT) to each window to obtain its spectrum. In our measurements, we use a $1ms$ window size[1] with 75% overlap between consecutive windows, which provides a balance between the computation complexity and frequency/time resolution. The rest of

[1] the window size should be determined based on sampling rate, clock frequency, and the required time resolution.

the training and monitoring operates on this sequence of spectra, where each spectrum (i.e., the spectrum of one window) is referred to as a Spectral Sample (SS).

5.2.2 Distance Metric for Comparing SSs

In both training and monitoring, REMOTE will need to compare SSs to each other, and for that, it requires a distance metric – a way to measure the "distance" between SSs in a way that corresponds how likely/unlikely they are to have been produced by execution of the same code. This distance metric should be sensitive to the aspects of the signal that change when executing different code, but insensitive to aspects of the signal that change between physical instances of the same device or over time on the same device instance. To achieve this, we create a new distance metric, ***Clock-Adjusted Energy and Peaks (CAPE)***.

Based on the insights from our prior work, the frequencies of the peaks in the signal around the clock frequency are an excellent foundation for constructing a distance metric that is sensitive to which region of code is executing. Unfortunately, our experiments have shown that the clock frequency can vary over time and among device instances, and a change in clock frequency also changes the frequencies of loop-related peaks around it. One difference is that, because the peaks' frequencies are all relative to the carrier frequency, any shift in the clock frequency also shifts the frequencies of the loop-related peaks by the same amount. The second change is caused by the relationship between clock frequency and program performance. Specifically, as the clock frequency increases, the program executes faster, leading to a lower per-iteration time T, higher frequency of the loop ($f_l = 1/T$), and thus moving the loop-related peaks away from the clock's frequency. Similarly, lower clock frequency moves the loop-related peaks closer to the clock frequency.

Thus the first step in computing our CAPE distance metric is to, for each frequency f that is of interest in an SS, compute the corresponding normalized frequency as $f_{norm} = \frac{f - f_{clk}}{f_{clk}}$, where f_{clk} is the clock frequency for that SS. This normalized frequency is expressed

as an offset from the clock frequency so that a shift in clock frequency does not change f_{norm} with it and is normalized to the clock, so it accounts for the clock frequency's first-order effect on execution time.

To make CAPE robust to weak signals and/or signals that have no well-defined peaks, we first consider the overall signal power (sum of magnitudes in the spectrum) of the signal outside the vicinity of the clock. The power of a poorly-defined peak is spread across a range of frequencies – visually it is a wide and not-very-tall "hump" rather than a narrow and tall "peak". When comparing two SSs that are different but each contain only "humps", if we only consider whether the SSs have power concentrations at the same (clock-adjusted) frequencies, the overlap among their "humps" causes these SSs to match much better than they otherwise should, and this can prevent detection of malware-induced changes in signals. Moreover, under poor signal-to-noise conditions (e.g., when the signal is received at a distance) sharp peaks are likely to still stand out of the noise, so due to random variation in noise, some "humps" end up below the noise level and some do not. For two SSs that should be the same (except for the noise), this causes poor matches, and this can lead to false positives. Thus to make our CAPE distance metric more robust against weak/noisy signals, we use a new insight, called ***"non-clock-energy"*** test, that non-clock energy varies very little among SSs that do belong to the same region, and that increases/decreases in a loop's overall per-iteration time concentrate less/more power toward the clock frequency in an SS. Therefore, SSs whose non-clock power differs by more than 0.5 dB are considered dissimilar by CAPE, and no further comparison between them is needed.

If the two SSs pass the "non-clock-energy" test, REMOTE compares them according to the (clock-adjusted) frequencies of their most prominent peaks. Specifically, we take N highest-magnitude frequency bins from the spectral sample (SS) that are each ***(i)*** not part of the *NoiseList*, and ***(ii)*** not within *D* spectral bins of a higher-amplitude spectral bin. The number N is determined differently for training and monitoring, as will be described shortly. The *NoiseList* contains frequencies of signals that are present regardless of which

specific region of the application is executing. For finding the *NoiseList*, we record the EM signal several times and average them while no program is being executed (the system is idle). We then choose 10 random SSs in the recorded signal, and then for each SS, sort it and find all the spikes that are at least 5dB above the noise floor and put them in the *NoiseList*. We empirically find that choosing 10 points is sufficient to find all the strong peaks since it can accurately capture the transient behavior of the environmental noise. It is also important to point that, using this method, our detection algorithm is robust to interference from nearby devices (that are not identical to the monitored device), as their clock and other frequently-occurring peaks will end up on the *NoiseList*. The reason for ignoring D spectral bins that are too close to even-higher-magnitude ones is that a very prominent peak in the spectrum typically has "slopes" whose magnitude can exceed the magnitude of other peaks, and we found that REMOTE is more robust when its decisions are based on separate peaks rather than just a few (possibly even one) very strong peaks and a number of frequency bins that belong to their "slopes".

Finally, REMOTE combines the information about the peaks' frequencies of two SSs into a single value that represents the distance among the SSs. For each peak in one SS, REMOTE finds the closest-frequency peak the other. If the frequency difference is large enough, the peak votes for a mismatch, and the ratio of the mismatch votes to the number of all (mismatch and match) votes is used as the distance metric between the two SSs.

5.2.3 Black-Box Training

To train REMOTE, signals are collected as the unmodified monitored device emanates them. However, care should be taken to achieve good coverage of the software behaviors, e.g., by using the same methods that are used to test program correctness. The problem of achieving good coverage tends to be easier for many applications in the CPS domain, especially those where correct operation is critical, because correctness concerns and the need for easy verification of correct operation motivates developers to produce code that has relatively

few code regions, and with very stable patterns for how the execution transitions between them. In such cases, normal use of the device is likely to provide good coverage of the application's code regions after a while.

After signals are obtained and converted into SSs, a key part of training is to associate SSs with the code regions they correspond to. To achieve this without using instrumentation or other on-the-monitored-device infrastructure, REMOTE relies on a general observation that a given region of code tends to produce EM signals whose SSs are similar to each other, while the SSs from different regions tend to differ from each other to various degrees. This observation allows us to group SSs according to similarity, and for that we use Hierarchical Density-Based Spatial Clustering of Applications with Noise **(HDBSCAN)**, a technique that performs clustering without any a priory knowledge about which cluster (region) each sample (SS) corresponds to, and with no a priori knowledge about the number of clusters (regions). Like other clustering algorithms, HDBCASN needs a distance metric, and in REMOTE that distance metric is the new CAPE metric defined in Section 5.2.2, using $N = 10$ peaks. Using this variation-robust metric allows training signals to be collected over time (e.g., over many hours), and/or on multiple device instances.

Because HDBSCAN clustering is based solely on similarity among SSs, its result may not precisely correspond to actual regions of the code, e.g., one region may produce more than one cluster if there are several distinct ways in which the region can execute, or two regions may end up in the same cluster if their execution produces very similar signals. Neither of these possibilities is a problem for REMOTE, and in fact, they result in improved sensitivity and performance. If separate clusters for distinct behaviors were forced into a single cluster, the resulting unified cluster would allow a wide variety of SSs to match - all the valid SS options and *also* everything that lays in-between in the distance-space used for clustering. By creating a separate cluster for each distinct possibility, REMOTE will detect anomalies that produce SSs that are not valid but lay in-between the valid ones. Conversely, when multiple regions are clustered together, they have very similar (practically indistin-

guishable) signals and it is more efficient and robust to treat them as one cluster. During monitoring, a Finite-State Machine (FSM) is used to keep track of the current region of the code. For each test, REMOTE compares the new SS to either the current region or any valid "next region" that has been seen during training.

5.2.4 Monitoring

During monitoring, REMOTE receives the signal and converts it to SSs in the same way it was done in training (same window size and overlap). After that, REMOTE can be viewed as a classifier that places each spectral sample (SS) into either one of the known categories (clusters identified during training) or into the "unknown" category that represents anomalous behavior, according to our CAPE distance metric (Figure 5.1 shows the flow-chart of REMOTE's monitoring algorithm). Specifically, a candidate region is rejected if its distance metric is above 50% (fewer than half of the peaks match). If all candidates are rejected, the observed SS is categorized as "unknown," otherwise it is categorized into the candidate category with the lowest CAPE distance metric. The number of peaks used for each cluster is no longer fixed at 10 – instead, it is identified for each cluster during training. We start with ten peaks, but then remove those that occur in fewer than 10% of the SS in the cluster. If this results in removing all peaks, we still retain the two most frequently occurring (among SSs from that cluster) peaks. This helps matching accuracy when the SSs in a cluster have few prominent peaks and a number of very weak peaks – in such cases it is more robust to use only the overall non-clock energy and the prominent peaks for matching than to use the peaks that may "disappear into the noise" due to changes in distance, antenna position, etc.

However, if the overall decision to report malware is based on only one SS, brief occurrences of strong transient noise can result in false positives. To avoid that, REMOTE only reports an anomaly if N consecutive SSs are classified as "unknown." The value of N should be chosen depending on the noise characteristics of the environment, but we found that N between 3 and 5 tends to work well in all our experiments. We use N=5

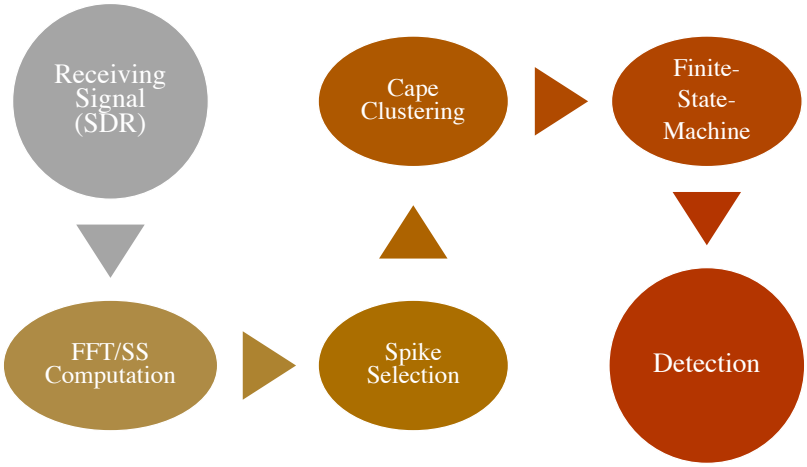


Figure 5.1: REMOTE's monitoring flow-chart.

because it biases REMOTE toward avoiding false positives, while still maintaining an excellent detection latency (N=5 corresponds to only 1.25 ms detection latency in our setup). As mentioned in Section 5.2.3, an FSM is used to count N.

Finally, we found that in the presence of an OS, interrupts and other system activity that occurs during an SS can make that SS dissimilar to those from training. For example, an interrupt that lasts $< 1ms$ can affect four consecutive SSs (recall that we use $1ms$ windows with 75% overlap), so a naive solution would be to add 4 to N (number of consecutive "unknown" SSs that are needed to trigger anomaly reporting). Using $N = 9$ indeed eliminates interrupt-induced false positives, but also prevents detection of attacks that are brief. Unfortunately, real-world malware (e.g., the attack on *Syringe-pump* that will be described in Section 5.3) can introduce only a short burst of activity into the otherwise-normal activity of the application. Fortunately, our experiments indicate that spectral features of interrupt activity are similar to each other, so during training interrupt activity can be isolated, and a separate set of clusters formed for various interrupts. During monitoring, REMOTE includes these clusters as candidates, allowing it to tolerate interrupts without becoming tolerant of similar-duration deviations from expected execution.

5.3 Experimental Results

5.3.1 Overview

In this section, we evaluate our framework using three set of experiments to show the effectiveness of REMOTE to detect different types of attack on variety of devices.

In the first set of experiments, we use two real-world CPS. The first CPS we use is an embedded medical device called *Syringe-pump* which is a representative of a medical cyber-physical system. The second system is a PID controller that is used for controlling the temperature of a soldering Iron. This type of system could also be used to control the temperature in other settings, such as a building or an industrial process, and thus is representative of a large class of industrial CPS/IoT systems.

For the second set of experiments, we use five applications from an embedded benchmark suite, MiBench [87], running on an IoT/embedded device, which are a representative of the computation that is needed in that market (e.g., automotive, industrial systems, etc.).

Finally, for the third part, we chose a robotic arm (LewanSoul LeArm 6DOF) [99], which is a representative of commonly-used CPS, currently existing in the market.

5.3.2 Measurement Setup

The measurement setup is shown in Figure 5.2. Depending on the distance, either a hand-made magnetic coil or a horn antenna is used to receive EM signals (no amplifier is used). For all measurements, we use a cheap ($<$\$30) software-defined radio (SDR) receiver (RTL-SDRv3) to record the signal. Using this radio, the entire cost for the near-field measurement setup (including the radio and a hand-made coil) is only around \$35, and for the far-field measurement setup is around \$100-200 (depending on the antenna). Further cost advantages can be gained if REMOTE is used in settings where multiple similar devices (with similar vulnerabilities) are used, so a single (or a few) devices can be monitored by REMOTE (especially in far-field scenario), with random changes to which specific devices are

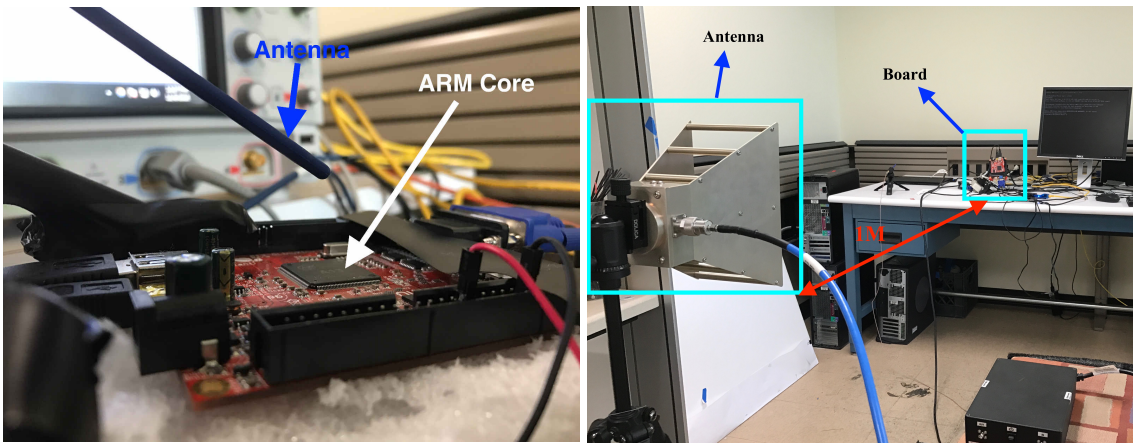


Figure 5.2: The near-field setup (left) consists of a small EM probe (PBS-E) [89], or a hand-made magnetic probe (not shown) placed 5 cm above the system's processor. A horn antenna placed 1 m away from the board for far-field measurements (right). In all cases, a software-defined radio (RTL-SDR [100]), priced around $30 and smaller and lighter than most portable USB hard drives, is used to record the signal.

monitored at any given time.

Note that all of our measurements were collected in the presence of other sources of the electromagnetic interface (EMI) including an active LCD that was intentionally placed about 15 cm behind the board. A set of TCL scripts are used to control the monitored system and the SDR (to record the signal). The entire REMOTE algorithm is implemented on a PC using Matlab2017-b.

5.3.3 File-less Attacks on Cyber-Physical-Systems

The part presents the results for two real-world CPS which are implemented on four different devices (shown in Table 5.1). To attack these devices we implement two end-to-end *file-less* attacks namely a ***code-reuse*** attack and an ***APT*** attack (advanced-persistent-threat).

Table 5.1: Boards used in this paper to evaluate REMOTE.

Device	Processor	Clock-rate	OS
Arduino Uno	ATMEGA-328p	16MHz	No
DE0-CV Altera FPGA	Nios-II softcore	50MHz	No
TS-7250	ARM9	200MHz	Debian
A13-OLinuXino	ARM Cortex A8	1GHz	Debian

The **first attack** implemented is a Code Reuse (CR) [101, 102] attack on a medical CPS called Syringe-pump. Syringe-pump is a medical device designed to dispense or withdraw a precise amount of fluid, e.g., in hospitals for applying medication at frequent interval [103].

The device typically consists of a syringe filled with medicine, an actuator (e.g., stepper motor), and a control unit that takes commands (e.g., amount of fluid to dispense/withdraw) and produces controls for the stepper motor. The systems must provide a high degree of reliability and assurance (typically by using a simple MAC) since imprecise or unwanted dispensing of medication, or failure to administer medication when needed can cause significant damage to the patient's health. In our evaluation, we use the Open Source Syringe Pump from [104] which also implemented in [105][2].

Our code-reuse attack involves overflowing the input buffer in reading the serial input function, which normally reads the input, sets a flag to indicate that new input is available, and returns. Exploiting this vulnerability, the return address in the stack is overwritten by a chain of gadget's addresses to launch an attack.

Since the security-critical part of this system is moving the syringe, Attacker's goal is being able to call the *MoveSyringe()* function, which is responsible for syringe movement, at an unwanted time while *skipping* the input checking part, *Delay()* function, which is responsible to check the authenticity of the command (otherwise the attacker needs to hack into the C&C server to send the commands which may not be a feasible task).

We use ROPGadget [106] for finding the proper chain of gadgets to put the address of *MoveSyringe()* in a register and branching to that function (from the *readInput()* function to skip the checking part).[3]. After branching to *MoveSyringe()* and executing it, PC jumps back to the main function and resumes normal behavior of the application.

Figure 5.3 shows a spectrogram of the Syringe-pump application in (top) malware-free run, and (bottom) when the CR attack happens. As seen in the figure, the Syringe-

[2]Note that the attack implemented in this paper is completely different that the one implemented in [105].
[3]For Nios-II we had to slightly change the ROPGadget to support the Nios-II ISA

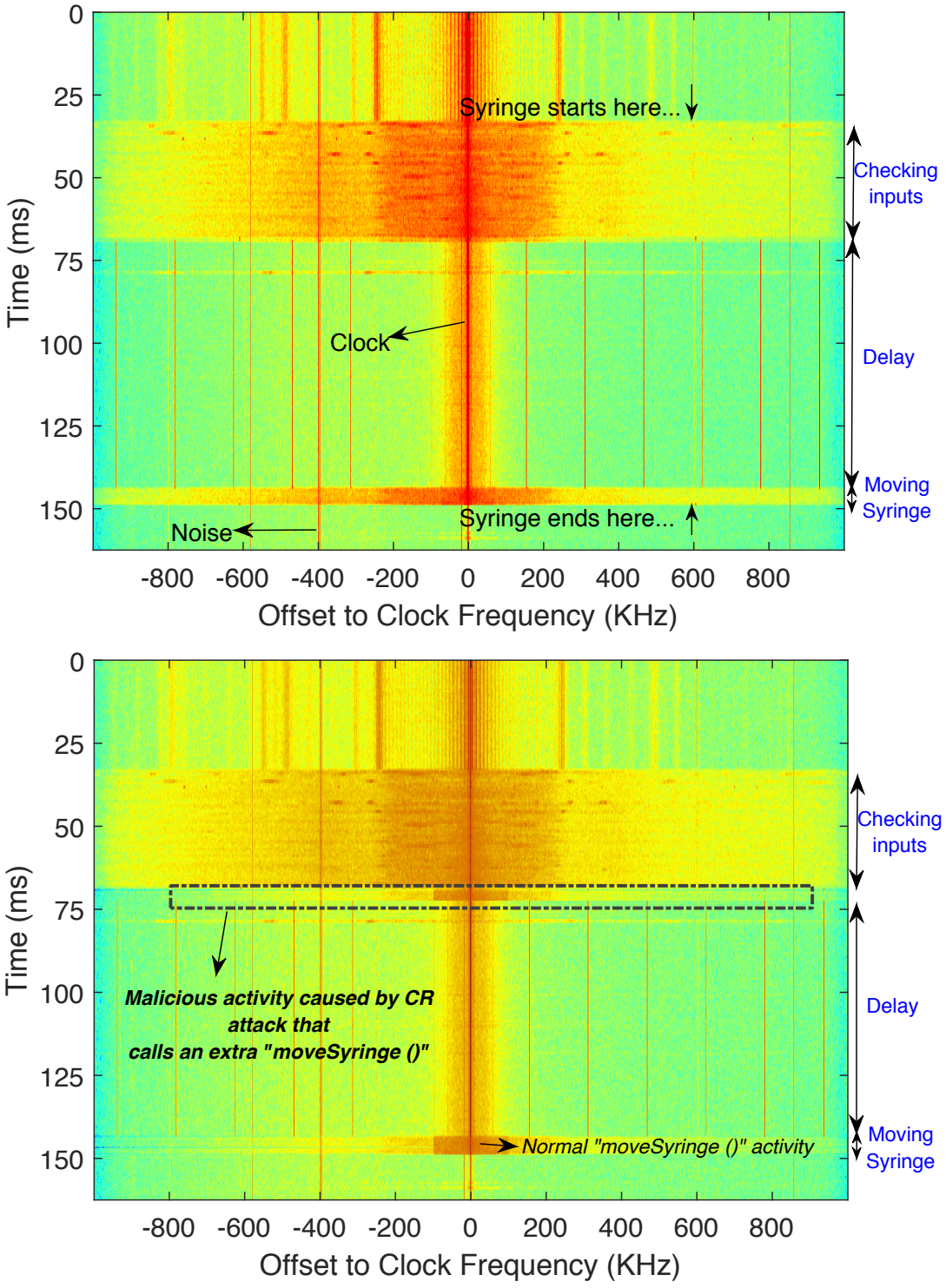


Figure 5.3: Spectrogram of the Syringe pump application in malware-free (top) and malware-afflicted (bottom) runs. Note that the differences in colors between the two spectrograms correspond to differences in signal magnitude which are caused by different positioning of the antenna. Such variation is common in practice and has almost no effect on REMOTE's functionality because REMOTE was designed to be robust to such variation.

pump application has three distinct regions with clearly different EM signatures: printing debug info and reading inputs, a delay/checker function which checks the message authenticity (using a simple MAC), and an actual movement of the syringe. The major difference between these two figures is the **reverse** order of "Delay" and *MoveSyringe()* parts in malicious run (bottom). In normal behavior, REMOTE expects to see $readInput \rightarrow Delay \rightarrow MoveSyirnge$ however, in CR attack, since the return address of the $readInput$ function is overwritten by the adversary, the code immediately jumps to *MoveSyringe()* and skips the "Delay" part, thus in the spectrogram, the third region (*MoveSyringe()*) is seen before "Delay" (bottom), which violates the correct ordering of regions and will be reported as "malicious" by REMOTE.

Our evaluation uses one attack per run in 25 runs, with REMOTE successfully detecting each of these attacks (see Table 5.2). We then performed 25 attack-free runs and found that REMOTE produced no false positives (see Table 5.2). To further evaluate our system and find more accurate results, we performed 1000 malware-free runs and 1000 malicious run on one device (Arduino)[4] for 24 hours. For these 2000 runs, REMOTE successfully found all the 1000 instances of malicious run and reported 997 out of 1000 malware-free runs as normal (i.e., only 3 out of 1000 false positive $= 0.3\%$).

Note that, depending on the size of injection, the *MoveSyringe()* in Syringe-pump could be very brief in time (e.g., around 3 ms as can be seen in Figure 5.3-left), and we found that without correctly handling the interrupts on Olimex and TS platforms (which have an operating system), we would either get very high false positives (due to interrupts), or high false negatives (by using large *N* to ignore short-term activity). However, as also discussed in subsection 5.2.4, by adding training-time samples for interrupts, we can use small *N*, while having 0% false positives.

Furthermore, we also repeated our measurement for Syringe-pump for both 50 cm and 1 m distances (using a 9 dBi horn antenna [107] connected to the SDR) and in both cases,

[4]To limit the amount of measurements and time for processing it, we picked only one of the four devices.

we also get perfect accuracy. It is also important to mention that the detection latency (i.e., the time attack starts until REMOTE detects it), for all four devices is $<$2 ms.

An alternative method for attacking Syringe-pump is by changing the *InjectionSize* (i.e., **Data-only attacks**). This also can be done using a CR attack. REMOTE is able to protect Syringe-pump against such attack since changing the *InjectionSize* will change the duration (i.e., the number of SSs) of *MoveSyringe()*. Since REMOTE is checking the signal in the granularity of SS, it can count the SSs which belong to *MoveSyringe()* activity and compare it to the expected number of SSs. To check how well REMOTE can detect such an attack, we check the number of SSs for *MoveSyringe()* for all the 25 attack-free runs and compare it to the actual *InjectionSize*. In all the instances, REMOTE reports the correct number of SSs. Note that we are not detecting EM emanations (RF) signal produced by the motor movement but the change in the code execution when "data-only" attack is performed. i.e., we observe the signal at clock frequency of the board and observe software changes, while motor movement signature occurs at much lower frequencies.

However, if the change is less than one SS or if the expected *InjectionSize* is unknown, REMOTE is not able to detect the change. Overall, there is a tradeoff between the size of SS and REMOTE's ability to detect small changes. Thus to improve the effectiveness of the system, either a higher sampling-rate setup can be used (smaller SS hence smaller detection granularity) or REMOTE can be combined with other existing methods (e.g., Data Confidentiality and Integrity (DCI) methods [108]) to protect the system against different types of data-only attacks. Finally, it is important to mention that However, as shown in this work (for this attack and other attacks in this section), meaningful attacks typically have much larger signature (i.e., order of milliseconds) than the current detection limit in REMOTE (200 microseconds).

The **second attack** is an advanced-persistent-threat (APT) attack on an industrial CPS (called Soldering-iron). A well-known example of such attack for CPS is Stuxnet. *Soldering-*

Table 5.2: Accuracy of REMOTE for several different systems and attack scenarios using various boards and applications.

Device (attack)	Syringe-pump (code-reuse attack)							
Board	*Arduino*		*Nios-II*		*TS-board*		*OLinuXino*	
Accuracy	True Pos.	False Pos.	True Pos.	False Pos.	True Pos.	False Pos.	True Pos.	False Pos.
	>99.9%	<0.3%	>99.9%	<0.1%	>99.9%	<0.1%	>99.9%	<0.1%

Device (attack)	Soldering-iron (APT attack)							
Board	*Arduino*		*Nios-II*		*TS-board*		*OLinuXino*	
Accuracy	True Pos.	False Pos.	True Pos.	False Pos.	True Pos.	False Pos.	True Pos.	False Pos.
	>99.9%	<0.1%	>99.9%	<0.1%	>99.9%	<0.1%	>99.9%	<0.1%

Device (attack)	Embedded/IoT (shellcode attack)									
App	*bitcount*		*basicmath*		*qsort*		*susan*		*fft*	
Accuracy	True Pos.	False Pos.	True Pos.	False Pos.	True Pos.	False Pos.	True Pos.	False Pos.	True Pos.	False Pos.
	>99.9%	<0.1%	>99.9%	<0.1%	>99.9%	<0.1%	100%	<0.1%	>99.9%	<0.1%

Device (attack)	Robotic-arm (firmware modification)	
Board	*LewanSoul LeArm 6DOF*	
Accuracy	True Pos.	False Pos.
	>98.2%	<0.2%

iron is an industrial CPS that allows users to specify a desired temperature for the iron and maintains it at that temperature using a proportional-integral-derivative (PID) controller. This type of controller could also be used to control the temperature in other settings, such as a building or an industrial process, and thus is representative of a large class of industrial CPS. This application is significantly larger than the Syringe-pump - with 70,000 instructions in its code and 1,020 static control-flow edges [104].

The application starts by initializing all the components. It then begins to control the Iron's temperature: it checks all the inputs (e.g., knob, push buttons, etc.) and then based on them decides to decrease or increase the temperature, prints new debug information on its display, etc. and then repeats this ad infinitum. The security-critical function is where the temperature of the iron is set *keepTemp()*. This function uses an iterative process (a PID controller) to change or keep the temperature of the iron. The critical variable is *temp_hist* – it holds the last two temperatures of the iron and is used to calculate the difference between the current temperature of the iron and these two last temperatures.

```
1 // The main loop
2 void loop() {
3     int16_t old_pos = read(&rotEncoder); // finding the position of the
    control knob
4     bool iron_on = isOn(&iron); // iron is the object for the soldering
    iron
5     // adding malicious activity
6   if (some_condition){
7     iron.temp_hist[0] = maliciousVal0;
```

```
8     iron.temp_hist[1] = maliciousVal1;
9     // where these values can be read from a file, memory or they could
      be random
10  } // end of malicious activity
11   byte bStatus = intButtonStatus(&rotButton); // reading input button
12   showScreen(pCurrentScreen);
13   keepTemp(&iron);    }
```

Example 5.1: A code fragment from the main loop of the soldering iron application and a possible injected malicious activity.

To implement a Stuxnet-like malware on this application, we assume that the attacker can reprogram the device. The attacker's goal is to change a critical value under some conditions, which in turn can cause damage to the overall system. A possible modification to the code is shown in Example 5.1 (lines 8-10), where based on one or several conditions (e.g., in our evaluation it checks the model of the device that is stored in memory), the temperature history can be changed. The key insight is that the added instructions will cause the spectral spikes during execution of the main loop to be shifted to lower frequencies (more time per iteration) as shown in Figure 5.4 for the A13-OLinuXino device.

To evaluate how well REMOTE can detect this type of attack, we use 7 runs for training and then use 25 runs without malware and 25 runs with malware to evaluate monitoring. Our results show REMOTE can successfully detect all the instances of the attack (a 100% true positive rate) (see Table 5.2).

5.3.4 Shellcode Attack on IoTs

Another popular class of attacks on CPS/IoTs are *shellcode* attacks where the adversary executes a malicious application (payload) through exploiting a software vulnerability. It is called "shellcode" because it typically starts a command shell (e.g., by executing *(/bin/sh)* binary) from which the attacker can control the compromised machine, but any piece of code that performs a similar task can be called shellcode. Once the attacker takes the

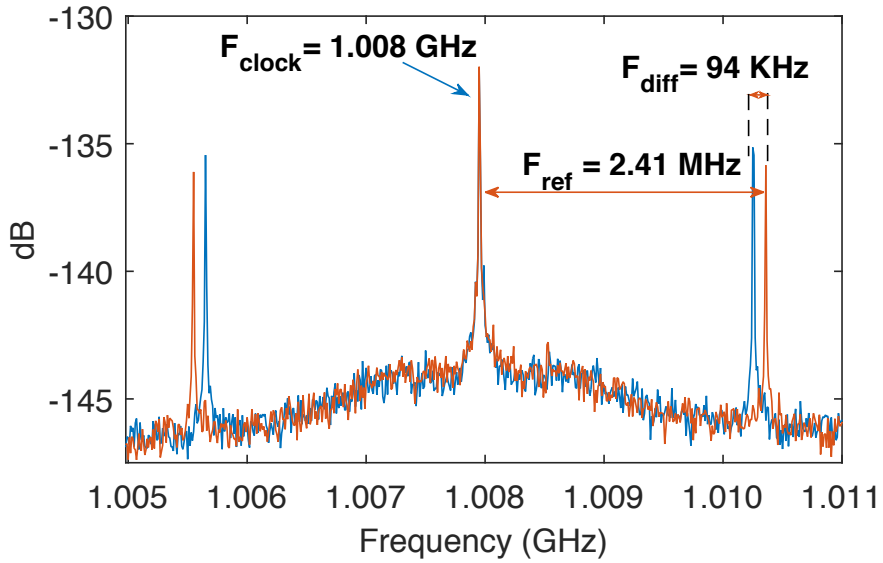


Figure 5.4: Adding malicious activity to the main loop of the Soldering-iron application (red: without malware, blue: with malware).

control, she can execute any *injected* code such as a *Denial-of-Service* attack.

In this paper, we implement this attack by invoking a shell *(/bin/sh)* via a buffer overflow exploit. We then run two malicious payloads on the invoked shell: a *DDoS bot*, and a *Ransomwmare*. These attacks typically target devices with operating systems. In this work, we implement them on an IoT device with an ARM core (A13-OLinuXino), which is a representative of state-of-the-art IoTs.

The attacks are implemented on five representative programs from MiBench suite (*bitcount*, *basicmath*, *qsort*, *susan*, and *fft*). We chose these applications among all the MiBench applications (this benchmark is designed to represent typical behaviors of embedded system: e.g., Security, Telecomm., Network, etc.) mainly because *bitcount* is a good representative of the applications that have several different distinct regions (our HDBSCAN clustering found 9 for this application) and has lots of different activities including nested-loops, recursive functions, interacting with memory, etc. *basicmath* is chosen because it is a good representative of unstable/weak activities since the activities in each region are very dependent on values (it is calculating different fundamental mathematics operations such as integration, square-root, etc.). We also chose *qsort* because it has lots of memory

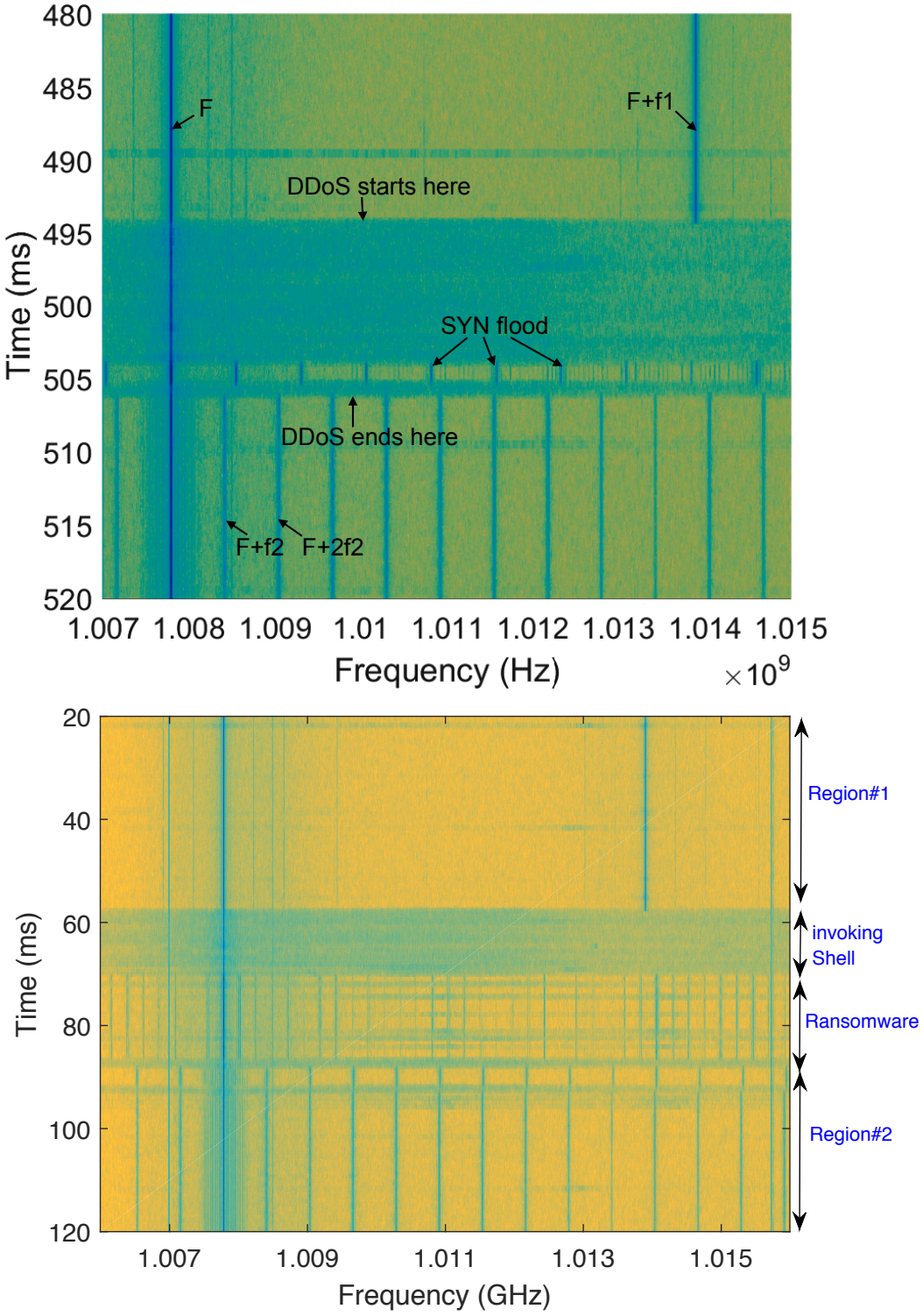


Figure 5.5: A run (top) where exploit, shellcode, and a 100-packet payload are injected into the execution between the original loops. A run (bottom) where exploit, shellcode, and a Ransomware payload are injected into the execution between the original loops.

accesses, and picked *susan* and *fft* since they are good representatives of common and popular activities in embedded system domain (i.e., image processing and telecomm.). In all these application, first a buffer-overflow vulnerability is exploited, and using a shellcode, a shell with same privileges as the original application is invoked. A malicious payload (i.e., DDoS or Ransomware) is then executed in this shell.

For the DDoS, we port the C&C and the bots from the Mirai open source to run on our IoT. The DDoS payload execution begins right after the shell is invoked and ends after sending 100 SYN packets. The application then resumes its normal activity. We use a PC on the local network as the target of the DDoS attack (SYN flood), and we verify on that PC that the attack is taking place. As another payload, we also implement a simple Ransomware prototype payload that uses AES-128 with CBC mode to encrypt data. This encryption represents the bulk of the execution activity created by Ransomware.

As in previous cases, we use 7 runs for training and then use 25 runs without malware and 25 runs with each malware (i.e., DDos and Ransomware) for all five applications. Our results (see Table 5.2) show REMOTE can successfully detect all the instances of the attack (a $>$99.9% true positive rate) while none of the malware-free runs incorrectly identified as malware (0% false positive rate).

We found that invoking a shell itself is visually detectable on our IoT device since it takes around 8 ms (about 32 SSs), and sending 100 SYN packets adds about 4 ms to that (see Fig. 5.5 (left) for DDoS and (right) for Ransomware).

5.3.5 APT Attack on Commercial CPS

The final system in our evaluation is a Robotic arm. It is often used for manufacturing and, typically, a critical component of any modern factory. It usually receives inputs/commands for a user and/or sensors and move objects based on these inputs. There is a growing concern in security of these CPSs since they are typically connected to the network and are exposed to cyber-threats [109]. In this work, we use a commercial robotic arm (LewanSoul

LeArm 6DOF [99]) which uses an Arduino board as a controller and a *Bluetooth* module to receive command. For this system, we implement an APT attack (firmware modification), where we assume that the reference libraries (e.g., library for Servo) are compromised (this can be also considered as a zero-day vulnerability). Note that, we assume that REMOTE 's training contains the "unmodified" version of these library (baseline reference data). In this attack, we modify a subroutine (*writeMicroseconds()*) in Arduino's Servo library [110] by adding an extra `if/else` condition to change the speed of Servo motor randomly and reprogram the system with this compromised library, assuming that the adversary is interested in causing a malfunction in arm's movement in real-time occasionally.

We use 7 runs for training and then use 1000 runs without and 1000 runs with the firmware modification. Our results (see Table 5.2) show REMOTE can successfully detect the instances of the attack with very high accuracy($>$98.2% true positive rate) while only less than 0.2% of the malware-free runs incorrectly identified as malware.

5.4 Further Evaluation of Robustness

5.4.1 Interrupts and System Activity

Among the platforms we tested, the longest-duration system activity "inserted" (via an interrupt) into the application activity tends to take a few milliseconds, and it appears to be associated with display management/update because disabling lightdm [111], the display manager, eliminates these interrupts (but other kinds of interrupts still occur). In contrast, in bare-metal devices interrupts (when there are any) tend to be around a microsecond in duration. Figure 5.6 shows the (perfect) ROC curve (solid blue line) for Syringe-pump on Olimex (and Debian Linux OS) when using REMOTE as described in section 5.2. We then prevented REMOTE from forming interrupt-activity clusters during training, and used the *EDDIE*'s scheme, and that has resulted in a severely degraded ROC curve (red dashed line) where many false positives are detected when 4 consecutive clusters are found to be "unknown" ($N = 4$ is subsection 5.2.4), and where increasing N reduces the false positives

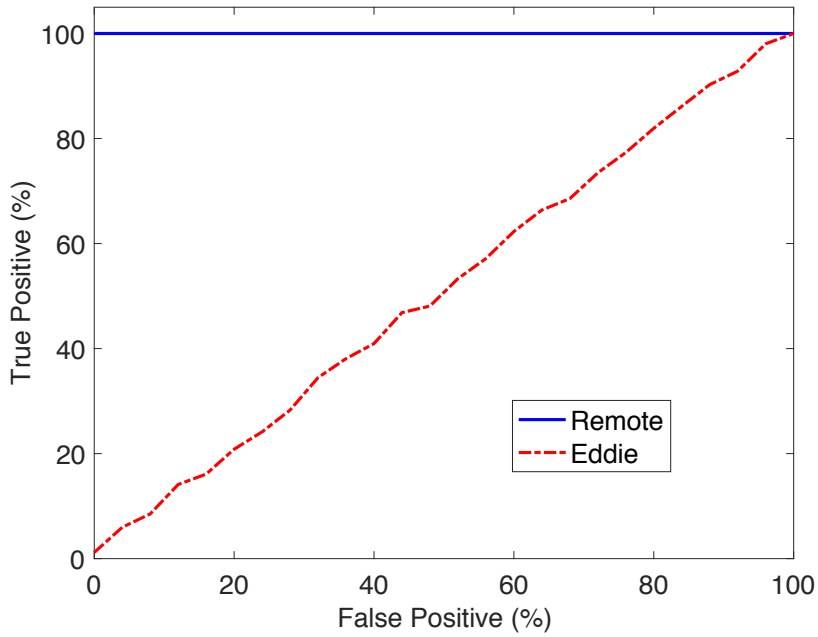


Figure 5.6: Accuracy of REMOTE with its mechanism for addressing interrupt activity (solid blue line) and *EDDIE* (red dashed line). The results are for the Syringe-pump software running on the Olimex board.

but also the true positives. This confirms that our approach of addressing system activity directly in REMOTE is significantly contributing to REMOTE's ability to detect malware while not reporting false positives due to system activity.

5.4.2 Hardware Platforms and Distance

Packaging and other limitations may require the EM signal to be received from some distance, which significantly weakens the signal. To evaluate the impact of distance on REMOTE, we receive the signal from distances of 5 cm, 50 cm, and 1 m away from each of the tested devices. To limit the amount of data that is recorded, we use only two representative programs from MiBench suite (*bitcount* and *basicmath*, described in subsection 5.3.4), and only two representative malware behaviors - one that adds a relatively small number of instructions inside a loop (*Stuxnet*-like), and another where similar malicious activity is done all-at-once outside of loops (*DDoS*-like).

For each device and each application, we use 25 malware-free runs and 25 runs for

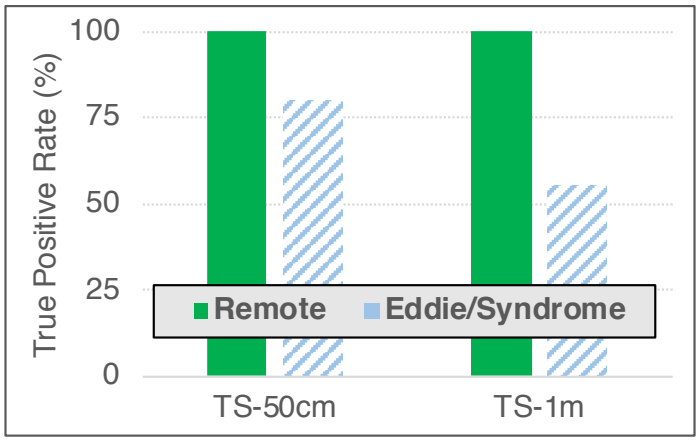


Figure 5.7: True positive rate (with 0% false positives) of REMOTE with its non-clock-power feature when comparing SSs (dark blue) and EDDIE/SYNDROME (light red). The results are for *basicmath* running on the TS board.

each of the two malware activities (75×3 runs for each of the platforms) to obtain the false negative (malware activity not reported in a malware-affected run) and false-positive rates (malware reported in a malware-free run) achieved by REMOTE. Our results show perfect accuracy (i.e., 0% false negatives and 0% false positives) for all devices and all three distances. However, if we prevent REMOTE from using total non-clock power when comparing SSs and use the scheme in *EDDIE* and/or *Syndrome*, on the TS board (which has the weakest signal among the boards tested) for 50 cm and 1 m distances we only observe 80% (at 50 cm) and 55% (at 1 m) true positive rates once we adjust other parameters to achieve 0% false positives (see Figure 5.7). This confirms that when signals are weak, comparisons based on spectral peaks alone are insufficient and other signal features (such as non-clock power used in REMOTE) must also be considered.

5.4.3 Manufacturing Variations

To study the effect of manufacturing variations on the EM signals and REMOTE accuracy, i.e., to determine if training is needed for each type of device or for each physical instance of a device, we use 30 physical instances of the Cyclone V DE0-CV Terrasic FPGA development board (chosen primarily because we have 30 such boards), to train REMOTE on one

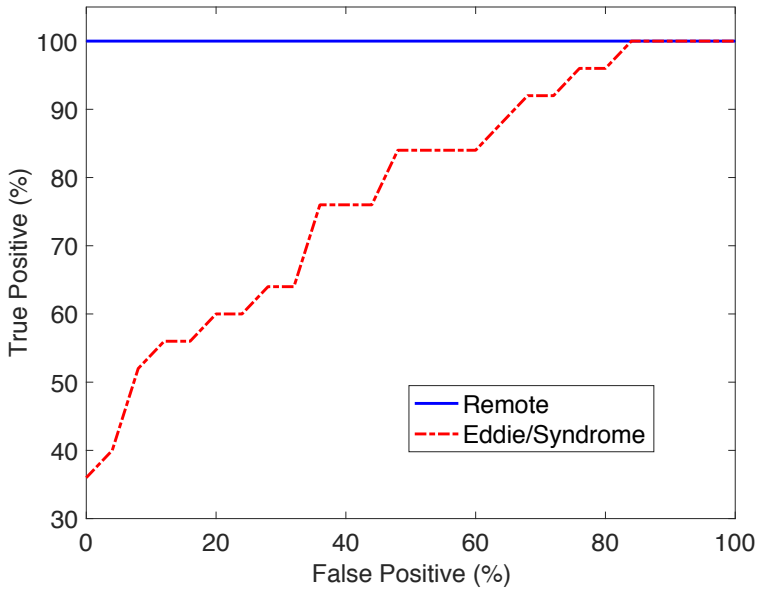


Figure 5.8: Accuracy for REMOTE with frequency-adjusting, vs. EDDIE/SYNDROME for FPGA board running *bitcount*.

board (randomly selected) and use that training to monitor each of the other 30 instances, with 20 runs of *bitcount* on each instance, both with and without malware.

Our results show that REMOTE's accuracy remains at 100% true positives and 0% false positives throughout this experiment. However, when we prevent REMOTE from frequency-adjusting the SSs used in comparisons, we still find no degradation for 17 of the boards, but for 13 the false positive rate increases to nearly 100%. Further analysis shows that the clock frequencies of the boards vary, with 17 of them (including the one trained-on) were within the frequency-tolerance (parameter D in subsection 5.2.2) of the matching, whereas the other 13 were outside the tolerance, causing none of their peaks to vote for the cluster the signal actually should belong to. If D is then adjusted to avoid false positives, the true positive rate is severely degraded. Figure 5.8 shows one such scenario where we trained on board number 3, and test on board number 4. The figure shows the ROC curve for board number 4 when frequency-adjusting is active and inactive. We also repeated this experiment for 10 Olimex boards (we do not have 30 of those), with very similar results with and without REMOTE's frequency-adjustment. These results confirm

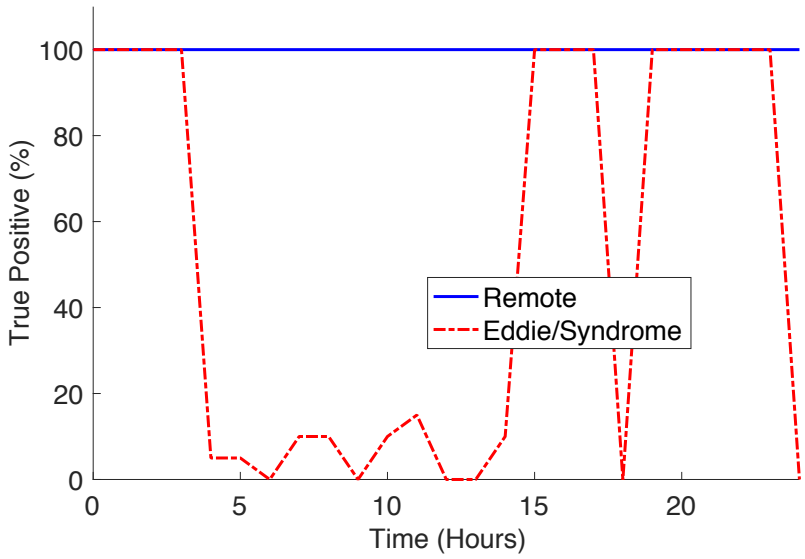


Figure 5.9: Performance of REMOTE with its clock-frequency adjustment feature vs. EDDIE/SYNDROME.

the need for frequency-adjustment in REMOTE if training and monitoring do not use the same physical instance of a device.

5.4.4 Variations Over Time

We record the signals at one-hour intervals, over a period of 24 hours, while keeping the FPGA board and the receiver active throughout the experiment, to observe how the emanated signals vary over time as device temperature (and room temperature) and external radio interference such as WiFi and cellular signals change during the day and due to the day/night transition. The set of measurements collected each hour consists of 60 *bitcount* runs, 20 without malware and 20 times with each of the two types of malware described in subsection 5.4.2. The training data for all REMOTE analyses in this experiment was recorded just after the device (FPGA board) and the receiver (SDR) were turned on.

We observed no deviation from REMOTE's accuracy (100% true positives and 0% false positives) throughout this experiment (solid blue line in Figure 5.9). We then prevent REMOTE from clock-adjusting the frequencies and repeat the experiments (on the same signal recordings), and find that the detection accuracy is dramatically degraded between hours 4

through 13 and hours 23 and 24 (dashed red in Figure 5.9). Further analysis shows that the clock frequency has shifted during these hours, coinciding with use of business-hours and off-hours thermostat setting for the room[5], likely because temperature affected the board's crystal oscillator whose signal is the basis for generating the processor's clock frequency.

5.4.5 Multi-Tasking/Time-Sharing

In our final set of experiments, we apply REMOTE in the runs where Ransomware (see Section 5.3.4) is executed as a separate process, without changing the application. The OLinuXino board only has one core, so its Debian Linux OS context-switches between the two processes until the Ransomware payload completes. Figure 5.10 shows the spectrogram in one such execution. In the first part of the spectrogram only the application is running. At some point (millisecond 812 in this spectrogram), the Ransomware process is started, and the context-switching in (approximately) 10 ms time-slices can clearly be seen beyond this point in the spectrogram. The spectrum of the malware process is clearly different from the spectrum produced by the application at this point in its execution, so we expect REMOTE to detect this malware execution scenario easily.

To quantitatively assess REMOTE detection for this scenario, we use 25 application runs, and in each run start the Ransomware process at a different point in the run. The results of this experiment are that REMOTE successfully detects all these runs even with the tolerance threshold that produces no false positives for malware-free executions, i.e., REMOTE produces an ideal ROC curve in this scenario. It should be noted here that in this set of runs, according to our threat model, the IoT system is running only one valid application. To successfully handle scenarios in which the system context-switches between multiple valid applications, REMOTE must be extended to identify when context switches are occurring and to keep track and validate spectral samples with the knowledge of which

[5]The actual change in clock frequency was less than one-part-per-million of the clock frequency, well within typical design tolerances for clock signals, and with negligible impact on the processor's overall performance and power consumption

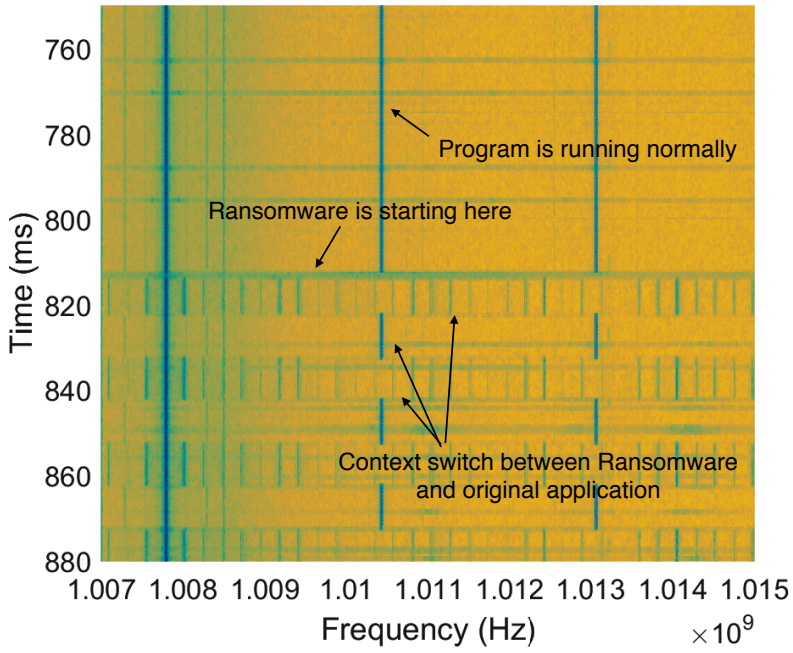


Figure 5.10: Spectrogram of context-switching between the unmodified *Bitcount* application and the Ransomware process.

application(s) they might belong and where the "current" point is in each of those applications. Although we believe such an extension to REMOTE is possible, it will likely require significant effort to figure out, implement, and evaluate, so we leave it for future work.

5.5 Prior Work and Practical Deployment

As described in Chapter 2, much work exists on using EM [63, 112, 113] and other physical signals [78, 114, 115] as side-channels for extracting sensitive information from a victim system. Majority of such work has focused on extracting keys during cryptographic activity, on countermeasures against such attacks and, more recently, on systematically identifying and quantifying EM signals that may be useful to attackers [116, 117], and on using EM/power analysis to identify the executed code at a per-instruction level [118, 119, 120, 121]. In addition to analog signals, contention on hardware resources such as caches [122, 123] that are shared between concurrently running threads or processes has been used as a side channel (extract information from a victim thread) and as a covert

channel (a thread secretly conveys sensitive information to another without using explicit communication mechanisms), and numerous countermeasures have been proposed ([124, 125]). However, as we described in this thesis, recent work (including work described in the earlier chapters) uses side-channel signals beneficially such as for profiling [126, 127], intrusion detection [**Nazari:2017**, 128, 105, 129], fingerprinting [130], characterization [131], etc. Callan *et al.* (ZOP [126]) performed profiling by matching the time-domain samples of the EM signal to program code at the granularity of acyclic paths[6]. The major limitation of the work is that it is very computationally demanding, so it is only practical for off-line analysis of relatively brief program runs.

Recently, Han *et al.* (ZEUS [128]) used spectral components of the signal as a feature, and an artificial neural network as a classifier, to detect a control-flow deviation in a PLC.

Comparing to the prior EM-based malware detectors that leveraged EM signals, REMOTE is specifically designed to be robust in the presence of, and evaluated for robustness to weak signals, poorly defined peaks in the spectrum, and variations in clock frequency among physical instances of the same device and over time on the same device. Furthermore, REMOTE is the first EM-based malware detector to be evaluated on real CPS applications affected by real malware prototypes (code-reuse attack, Stuxnet-like, and shellcode), and also the first to be evaluated on several platforms, with very different processor architectures, including platforms that run the application on bare-metal and those with an OS. Moreover, it is first to experimentally identify the specific ways in which software activity, distance, enclosure, and antenna positioning, and variation over time and among device instances of the same time, affect the received signal.

Another related body of work are power consumption-based malware detection frameworks. Liu *et al.* [47] provide code execution tracking based on the power signal using an HMM model to recover most likely executed instruction sequence with a revised Viterbi algorithm. Kim *et al.* [132] build signatures for individual pieces of malware and later

[6] An acyclic path typically contains one or a few basic blocks.

recognize them. Clark *et al.* (*WattsUpDoc* [83]) detect malware by monitoring system-wide power consumption. Liu *et al.* (VirusMeter [133]) build a state machine and flags later anomalous power consumption using a variety of classifiers. Gonzalez *et al.* (Power Fingerprinting [134]) demonstrate that a specific function and its modified version can be distinguished using LDA/PCA. Compared to these works, REMOTE has the following advantages: *(i)* unlike these methods, REMOTE does not require access to the source code for training thus it's more suitable for scenarios where the code is not available (propitiatory) and/or the existing infrastructure on the device does not support instrumentation. *(ii)* our evaluations show that REMOTE is applicable to a variety of systems with various software/hardware designs. Unfortunately, existing work [47, 132, 133, 134] provide very limited or no study on the robustness of their approach which makes it difficult to fairly judge their scalability and usefulness on other platforms. Further, it is also unknown how these systems behave (in therms of accuracy) under different sources of variations. Since as shown in section 5.4, these variations may have significant impact on the overall accuracy. *(iii)* these methods often suffer from non-negligible False-Positive-Rate (e.g., 3% [132], 2% [83], 5% [133]) which can significantly degrade the usefulness of the intrusion detector as an on-line, always-on protection system. Finally, *(iv)* instead of leveraging the power consumption as a side-channel, REMOTE uses the EM signal that provides locality, higher bandwidth, and requires no physical connection to the monitored device.

We believe that the main advantages of using REMOTE in industrial CPS, as shown in Section 5.4, are that *(a)* REMOTE can be trained on one device and then monitor other devices of the same kind, *(b)* REMOTE is robust to the presence of EM noise and interference from other (non-monitored) electronic devices, *(c)* REMOTE can be used at a distance and without making any changes to the heavy-duty plastic enclosures that are typically used in such settings, and without opening the enclosure (which often voids the manufacturer's warranty) and *(d)* REMOTE does not require access to source code which is usually unavailable (proprietary).

5.6 Conclusions

In this chapter we proposed REMOTE, a new robust framework to detect malware by externally observing EM signals emitted by a CPS. REMOTE does not require any resources or infrastructure on, or any modifications to, the monitored system itself, which makes it especially suitable for malware detection on CPS where hardware resources may be limited and performance and energy overheads introduced by other monitoring approaches may be unacceptable. REMOTE can identify malicious code injection into a known application that is running on a CPS in real time and with a low detection latency.

To develop a robust framework, we systematically explored practical concerns through experiments and analysis. First, to demonstrate the usability of REMOTE in real-world scenarios, we ported several real-world cyber-physical-systems each with a meaningful attack, to different platforms. Our results showed that for all of the programs on each of the platforms, REMOTE successfully detected the instances of attacks with high accuracy and almost no false positives. We then systematically evaluated the robustness of REMOTE to interrupts and other system activity, to signal variation among different physical instances of the same device, to changes in antenna distance, and to changes over time. By selectively disabling the robustness-oriented features of REMOTE, we also demonstrated that these features are indeed contributing to its robustness.

Using these measurements and analysis, we showed REMOTE has several advantages over state-of-the-art external malware detection frameworks and it is a promising candidate for protecting CPS when implementing an internal malware detector is infeasible.

CHAPTER 6

EMMA: HARDWARE/SOFTWARE ATTESTATION FRAMEWORK FOR EMBEDDED SYSTEMS USING ELECTROMAGNETIC SIGNALS

6.1 Abstract

Establishing trust for an execution environment is an important problem, and practical solutions for it rely on *attestation*, where a potentially untrusted system (prover) computes a response to a challenge sent by the trusted system (verifier). The response computation typically involves measurement (e.g., a checksum) of the prover's program code and execution environment, which the verifier checks against expected values for a "clean" (trustworthy) system. The main challenge in attestation is that, in addition to checking the response, the verifier also needs to verify the integrity of the response computation itself, i.e., that response computation itself has not been tampered with to produce expected values without measuring the verifier's actual code and environment.

On higher-end processors, this integrity of response computation is verified cryptographically, using *dedicated* trusted hardware (e.g., SGX). On embedded systems, however, form factor, battery life, and other constraints prevent the use of such sophisticated hardware support. Instead, a popular approach is to use the *request-to-response* time as a way to establish some level of confidence about the integrity of the response computation itself. However, the overall request-to-response time provides only one coarse-grained measurement from which the integrity of the attestation is to be inferred, and even this one measurement is noisy because it includes the round-trip network latency and/or variations due to micro-architectural events. Thus, the attestation is vulnerable to attacks where the adversary has tampered with response computation, but the resulting additional computation time is small relative to the overall request-to-response time.

To develop an effective software attestation framework, we make a key observation that the existing approach of execution-time measurement for attestation is only one example of using externally measurable side-channel information and that other side channels, some of which can provide much finer-grain information about the response computation, can be used. As a proof of concept, we propose EMMA, a novel method for attestation that leverages electromagnetic side-channel signals that are emanated by the embedded system during response computation, to confirm that the embedded device has, upon receiving the challenge, actually computed the response using the valid program code for that computation. This new approach requires physical proximity, but imposes no overhead to the system, and provides highly accurate monitoring *during* the attestation process. We implement EMMA on a popular embedded system, Arduino UNO, and evaluate our system with a wide range of attacks on attestation integrity, including *memory-copy*, *return-oriented rootkit*, and *proxy* attacks. Our results show that EMMA can successfully detect each instance of these attacks, with no false positives, in contrast to existing methods where many of the attacks succeed without detection. Further, we show how EMMA can be scaled to attest multiple devices simultaneously and/or systems with different and more complex architectures, and demonstrate its robustness against different sources of variability.

Followings are the contributions of this work:

- A new attestation method based on electromagnetic emanations of the prover,
- A proof-of-concept implementation of this attestation method, which we call EMMA,
- Evaluation of EMMA on five different types of attacks,
- Further analysis on EMMA for its applicability on other platforms and its robustness against different sources of variations (e.g., distance, time, micro-architecture, etc.).

6.2 Background

6.2.1 Attestation

Attestation is a security primitive that allows a trusted system (verifier) to verify the integrity of program code, execution environment, data values, etc. in a potentially untrusted system (prover). Attestation typically relies on a *challenge-response* paradigm, where the prover is asked to calculate a checksum over verifier-requested parts of a program/-data memory contents. The response computation typically involves measurement (e.g., a checksum) of the prover's program code and execution environment, which the verifier checks against expected values for a "clean" (trustworthy) system. The verifier considers the prover's integrity to not be compromised if *(i)* the checksum provided by the prover matched with the *expected* value computed by the verifier, and *(ii)* the computation that produced the response itself has not been tampered with, e.g., to falsify the expected values without actually computing them from the verifier's actual code and data.

In high-end processors, the assurance that the response computation itself was not tampered with is typically provided by using a hardware-supported Trusted Execution Environment (TEE), which uses dedicated hardware (e.g., SGX, TPM, etc.) within the prover.

On embedded systems, however, form factor, battery life, and other constraints prevent the use of hardware-supported enclaves or other sophisticated hardware support. Instead, a popular approach, *Software Attestation* ([135, 136, 137, 138, 139, 140]), is to compute the checksum in software, using ordinary execution on the prover, and to leverage measurement of the request-to-response time as a way to establish some level of confidence about the integrity of the response computation itself.

To implement this method, the verifier utilizes the challenge-response paradigm by asking the embedded system (prover) to compute a checksum of its program memory, while measuring the response time to prevent the adversary from computing a correct response, e.g., by temporarily restoring the program memory while the response is computed, by

using a copy of the program data to compute the response, by forwarding the challenge to another system that computes the response, etc. The prover passes the attestation test only if it provides the correct response to the challenge (i.e., $Response_{prover} = Response_{expected}$) without violating the timing requirement (i.e., $t_{response} < t_{threshold}$).

Unfortunately, the overall request-to-response time provides only one coarse-grained measurement, and this method is not able to monitor the prover *during* the attestation procedure without imposing a significant performance and/or cost overhead to the system. This, in turn, makes the software attestation schemes vulnerable to attacks which have very low-latency compared to the overall response time (i.e., $t_{attack} << t_{threshold}$). Moreover, due to the network limitations and/or micro-architectural events, this request-to-response time may be noisy since it includes the round-trip network latency and/or variations caused by the micro-architectural events (e.g., cache miss) which consequently, causes a further increase in $t_{threshold}$ (to tolerate the variance and reduce the false positive rate), and hence, potentially makes these schemes even more vulnerable to low-latency attacks.

To address this limitation, in this work, by analyzing EM side-channel signals in the frequency domain, we develop an EM-Monitoring algorithm that can *(a)* infer the time point when attestation activity begins and ends in the prover by checking when the spikes correspond to the checksum loop appear and disappear, *(b)* check whether the attestation process matches with a known-good model (to ensure that this process is not modified by an adversary) by checking the frequency of the spikes.

6.2.2 Threat Model and Assumptions

We assume that the adversary has installed malicious code on the target embedded system, with full control over the hardware and software of the device, including the ability to arbitrarily modify program and data memory, or any other memories available on the device. The attack succeeds if the device passes the attestation despite the presence of a malicious code. Note that attestation does not depend on how malicious code was origi-

nally installed on the device, and methods for doing so are abundantly represented in the research literature, although we do not discuss them in detail in this chapter.

Unlike most prior software-based methods, we assume that the prover (attested device) *can send messages* to and collude with other *faster* malicious peers, e.g., to use them as proxies for calculating the checksum faster. Finally, we assume that the verifier has full information about the prover's architecture (e.g., address space, memory architecture, etc.).

6.3 Design Overview

An overview of the EMMA framework is shown in Figure 6.1. This framework consists of a Verifier, $\mathcal{V}$, (e.g., a trusted PC) and a Prover, $\mathcal{P}$, (e.g., an embedded system). The Verifier includes, or is connected to, a monitoring system (EM-MON) that can receive and analyze the EM signals unintentionally emanated by $\mathcal{P}$. Attestation begins with $\mathcal{V}$ preparing a challenge locally (❶). The challenge includes a seed value (which will be used later to initialize a Pseudo-Random Number Generator (PRNG) in $\mathcal{P}$), an address range, the total number of iterations for checksum loop, a random value to initialize the checksum in $\mathcal{P}$, and a random nonce. The challenge is then sent to the $\mathcal{P}$ via a communication link (❷) which invokes the *verification function*, `attest()`, and starts the "attestation procedure" by causing an interrupt on $\mathcal{P}$. Note that this function runs at the highest prover's processor privilege level with interrupts turned off. In software attestation, the dynamic root of trust is instantiated through the *verification function*, a self-checking function that computes a checksum over its own instructions and sends it to the verifier.

Upon sending the challenge to $\mathcal{P}$, $\mathcal{V}$ also starts the "monitoring process" (❸) on EM-MON. Through analysis of EM signals emanated from $\mathcal{P}$, EM-MON determines three critical values about $\mathcal{P}$'s checksum computation and reports an error if any of them deviates from a known-good model (*reference* model). These tasks/values include *(i)* the delay between reception of the challenge and start of the checksum loop (initialization phase). *(ii)* the checksum loop's EM signature (i.e., frequency of spikes). *(iii)* the total attestation

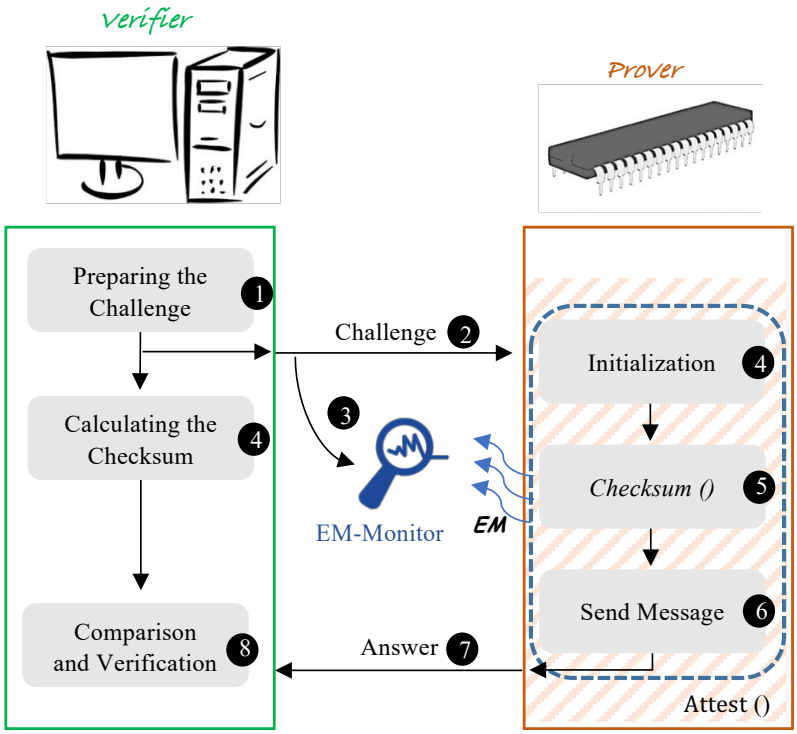


Figure 6.1: Overview of EMMA framework.

time on $\mathcal{V}$. These values/tasks are primarily chosen because fundamentally, there are two critical durations in the execution of the attestation process on the prover: *(a)* the time that is taken between receipt of the command to perform attestation and the start of the actual checksum process (because an adversary could contact another device during this period, for the proxy attack, or quickly hide the malicious code before starting the procedure), and *(b)* the time taken for the checksum process itself (because an adversary could try to do "extra work" during checksumming to hide the malicious code).

After sending the challenge, the verifier, independently, calculates the "expected" checksum on its own (known-good) copy of the embedded system's program memory (❹). At the same time, the self-checking verification function starts with initializing its local variables based on the received challenge (❹), and then it starts the "checksum calculation", `Checksum()` (❺). This function is an optimized loop which, in each iteration, reads a memory line in a pseudo-random fashion, and updates the checksum based on the content of that address. The address range and the total number of iterations of the loop are

determined by the challenge.

Once `Checksum()` is finished, $\mathcal{P}$ forms a response (❻) that includes the calculated checksum and the random nonce which initially was sent by $\mathcal{V}$, and sends this response to $\mathcal{V}$ (❼). The original nonce acts as an identification and helps to increase the overhead for a proxy attack. Finally, $\mathcal{V}$ compares the information received from $\mathcal{P}$ with its pre-computed checksum and the original challenge and compares the results from EM-MON to the expected ones. If they all match, one trial of attestation will finish successfully (❽).

At this point the *dynamic root of trust* has been established; thus, the verification function can either invoke an executable, and hence, provides a TEE, or invoke a *hash* computation function to compute the hash value over the prover's memory contents (entirely or partially). This hash value can then be sent back, which in turn, provides the current state of the prover to the verifier. Note that all of these functionalities are still part of the verification function; thus, they have been used in computing the checksum, which means it is guaranteed that the hash function or invoking the executable is also untampered with.

6.3.1 Verification Function

This function has three main phases: a *prologue* which is responsible to initialize some values based on the received challenge, *checksum computation*, and an *epilogue* which is responsible for sending the response back to the verifier and invoking the executable or hash computation function. It is important to point out that almost all of the execution time is spent in the checksum computation phase, where the prover repeatedly reads different lines of its memory and updates the checksum based on that.

To attack this function, the adversary has two options: she can either modify the checksum calculation function (e.g., to change the requested addresses) or modify the prologue/epilogue phases (e.g., to forward the challenge to another device). The main challenge for the adversary is that while changing the checksum calculation is desired and more effective, any change (even single instruction) in the checksum computation phase will be

significantly magnified due to a large number of iterations of the checksum loop. Thus, the adversary will be faced with a fundamental choice between having either a large but short-term malicious activity in the epilogue/prologue, or alternatively, a small, but long-term malicious activity in the checksum phase. Hence, to detect attacks on the verification function, an ideal detection framework should be able to detect single-instruction modifications in the checksum loop, and tiny[1] changes in the epilogue and/or prologue phases.

Knowing these facts, the attestation procedure has to be designed carefully so that it significantly degrades the ability of the malicious code to remain undetected during attestation. To achieve such a robust attestation procedure, there are several important factors that need to be considered. In this paper, we will briefly mention the key factors, but we refer the readers to these prior proposals [137, 138, 139, 140, 141] for more details on why these factors are important, and how they provide a strong security guarantee. Particularly, we chose the method used in [140] due to its simplicity and robustness. Following briefly overviews the important factors considered in designing such a secure checksum computation function:

- The checksum computed by the device should be a function of the challenge sent by the verifier to prevent pre-computation and replay attacks.
- To prevent the adversary from predicting and redirecting the memory requests, the addresses should be generated in a pseudo-random fashion.
- The checksum function should be *strongly-ordered*. A strongly-ordered function is a function whose output differs with high probability if the operations are evaluated in a different order. A strongly-ordered function requires an adversary to perform the same operations on the same data in the same sequence as the original function to obtain the correct result.
- The checksum function should have a low variance execution time so that the adversary can not exploit the variation to her advantage. Furthermore, the code should be optimized

[1] the size of this detection depends on how small a meaningful attack could be.

Algorithm 2 The checksum computation algorithm used in EMMA.

1: Initialization:
2: $PRNG = seed$
3: Set $MASK$ based on $beginAddress$ and $endAddress$
4: $memOffset = beginAddr$
5: $cSum = seed$
6: Checksum: *// checksum main loop* `(Checksum())`
7: **for** i=1 to $totIter$ **do**
8: **for** j=1 to 10 **do**
9: $PRNG = PRNG + (PRNG^2 \vee 5)\ mod\ 2^{16}$
10: $memAddr = memAddr \oplus PRNG$
11: $memAddr = (memAddr \wedge MASK) + memOffset$
12: $cSum_j = cSum_j + (Mem[memAddr] \oplus cSum_{j-1})$
13: $cSum_j = cSum_j + (i \oplus PC)$
14: $cSum_j = cSum_j + (PRNG \oplus memAddr)$
15: $cSum_j = cSum_j + (SR \oplus cSum_{j-2})$
16: $cSum_j = rotateLeft(cSum_j)$
17: **end for**
18: **end for**

and non-parallelizable.

Algorithm 2 shows the pseudo-code for our checksum computation based on the scheme proposed in [140]. The main checksum loop consists of a series of alternating XOR and ADD instructions. This series has the property of being strongly-ordered [141]; thus, none of the operations can be re-ordered or removed. Furthermore, using this sequence prevents parallelization and out-of-order execution since, at any step, the current value is needed to compute the succeeding values. As suggested in [140], we use a 160-bit long checksum to keep all the registers busy and to significantly reduce the collision probability. The checksum is stored as a vector in a set of 8/16-bit general purpose registers (blocks) depending on the architecture of the processor (i.e., AVR, ARM, etc.). As mentioned earlier, memory is traversed in a pseudo-random fashion by using a PRNG. Similar to previous work, we use a 16-bit T-function [142] to generate these random numbers. Each partial checksum block is also dependent on *(a)* the last two calculated partial sums; to avoid parallelization and pre-computation attack, *(b)* a key; to avoid replay attack, *(c)* current memory address (data pointer) and PC (if available depending on the architecture); to avoid memory copy

attack, *(d)* the content of the program memory; to avoid changing the attestation code, and *(e)* the Status Register (SR) to check the status of the interrupt-disable flag.

Note that our checksum loop is designed such that it is not vulnerable to the attack introduced by Castelluccia *et al.* [143], where changing PC and $memAddr$ simultaneously will result in the same checksum. To optimize the performance and avoid possible branch mis-predictions, in the actual implementation of the checksum, the inner loop is unrolled to calculate all the partial sums in one iteration. The detailed implementation of this code on an Arduino Uno will be shown in Section 6.4.

6.3.2 Security Analysis

The adversary model in software attestation is fundamentally different from classical cryptographic adversary models. Typically, the adversary is modeled by a polynomially bounded algorithm that aims to achieve a certain goal without having certain knowledge (e.g., cryptographic keys). In contrast, an adversary against a software attestation scheme can be unbounded in principle, and has complete knowledge of the prover device configuration and state. However, during the attack, it has to specify (or program) a malicious prover device with tight resource constraints. In other words, the adversary has unbound resources for preparing the attack but only a tight time-bound and limited computational and memory resources for executing the attack.

Due to these differences, and the fact that software attestation methods cannot rely on any trusted secret on the prover (or verifier), software attestation follows a fundamentally different approach, and leverages time side-channel information. A basic requirement of this approach is that the verifier, $\mathcal{V}$, specifies a practically optimal implementation of the algorithm that processes the challenge according to the attestation algorithm. This means that it should be hard to find any other implementation of this algorithm that can be executed by a prover, $\mathcal{P}$, in significantly less time than $t_{threshold}$. Otherwise, a malicious prover could use a faster implementation and exploit the time difference to perform additional

computations, e.g., to lie about its state. Hence, to ensure that $\mathcal{V}$ can measure also slight changes to the prover's code, $\mathcal{V}$ needs to amplify the effect of such changes. The most promising approach to realize this in practice is designing the attestation protocol as an iterative algorithm with a large number of rounds.

The core components of any software attestation algorithm are the address generation algorithm (Gen) and the checksum calculation/compression scheme (ChK). To prevent a malicious prover, $\widetilde{\mathcal{P}}$, from using pre-computed attestation responses, the memory addresses a_i generated by the random address generator, Gen, should be "sufficiently random" [144]. Ideally, Gen should be able to generate any number for $(a_1, ..., a_N)$, where N is the maximum achievable number for an m-bit address space. While this is impossible from an information-theoretic point of view, the best one may ask for is that the memory addresses a_i generated by Gen should be *computationally indistinguishable* from uniformly random values within a certain time-bound, t, (assuming that $\widetilde{\mathcal{P}}$ does not know the seed in advance).

In principle, nothing prevents $\widetilde{\mathcal{P}}$ from using an arbitrary seed value to compute all the possible addresses (i.e., $(a_1, ..., a_N)$) on its own, making them easily distinguishable from random values. The best can be done is to require that $\widetilde{\mathcal{P}}$ cannot derive any meaningful information about a_{i+1} from a_i and the seed without investing a certain minimum amount of time. Specifically, we assume that an algorithm with input s that does not execute Gen cannot distinguish $a_{i+1} = Gen(a_i)$ from uniformly random values. This property holds true for the T-functions as shown in [142], since either the adversary needs to spend the same amount of time as Gen to compute the next address or save all possible addresses, and read them one by one when necessary. However, the latter ends up taking much more time than the former since for a 16-bit T-function, the adversary needs to save more than 128KB of data which clearly doesn't fit in an L1 cache; thus, the adversary needs to pay the L2 (or main memory) delay penalty, which is significantly larger than computing the random value using T-function. In short, the size and bijection property of the T-function used in our algorithm satisfies the security requirements for Gen.

The purpose of the checksum function, ChK, is to map the memory state of the prover, $\mathcal{P}$, to a smaller attestation response, r, which reduces the amount of data to be sent from $\mathcal{P}$ to the verifier $\mathcal{V}$. Note that the output of ChK depends also on the challenge sent by the verifier to avoid replay and/or pre-computation attacks. A necessary security requirement on Chk is that it should be *hard* for a malicious prover, $\widetilde{\mathcal{P}}$, to replace the correct input, S, to Chk with some other value, $\widetilde{S} \neq S$, that yields the same attestation response r. This is similar to the common notion of *second pre-image resistance* of cryptographic hash functions. However, due to the time-bound property of the software attestation scheme, it is sufficient that ChK fulfills only a *much weaker* form of the second pre-image resistance since we need to consider only "blind" adversaries who (in contrast to the classical definition of second pre-image resistance) do not even know the correct response to the verifier's challenge. The reason is that, as soon as $\widetilde{\mathcal{P}}$ knows the correct response, he could send it to $\mathcal{V}$, and would not bother to determine a second pre-image.

Using this fact, our checksum is chosen large enough (160-bit) to significantly reduce the chance of the collision and make it computationally hard for a second pre-image attack. This can be proven, as shown in [145], that ChK used in this paper provides an almost full coverage (i.e., almost all possible numbers in $[0, 2^{16})$ for a 16-bit partial checksum), which, in turn, makes ChK resistant to blind pre-image attacks. As a result, to break the checksum function, the adversary can either remember all possible challenge-response pairs (called "challenge-buffering" attack) or modify the checksum calculation function in real-time to produce the correct response even in the presence of the malicious code.

In our setup, implementing the "challenge-buffering" attack is not feasible in practice due to the significantly large size of the input space (i.e., $> 2^{32}$ possibilities). Furthermore, as extensively shown in this paper, any attempt in changing the checksum loop will cause a shift in the loop's spectral signature, and hence, will be detected by EMMA. Thus, using EMMA will make both Gen and ChK components, and hence the entire system cryptographically secure against cyber-attacks.

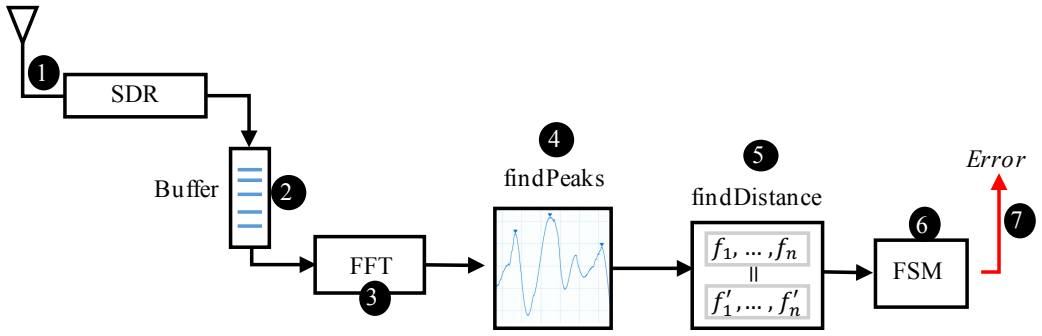


Figure 6.2: Overview of EM monitoring framework.

6.3.3 EM-Monitoring

The EM-MON component ensures that the attestation computation in $\mathcal{V}$ was not tampered with. Figure 6.2 shows this monitoring framework. Using an antenna (e.g., a magnetic probe) and a signal acquisition device (e.g., a software-defined-radio), the EM signal is captured and received as a time-series (❶). The signal is then transformed into a sequence of Short *Frequency-Domain* Samples (SFDS) using a Short-Time Fourier Transform (STFT). This transformation consists of dividing the EM signals into consecutive, equal-length, and overlapping segments of size t and then computing the STFT from each of these segments to obtain the corresponding SFDS (❷ and ❸). Segment size, t, has to be chosen such that it provides a balance between the time resolution and EM-MON's computational needs. Also, t should be long enough to capture several iterations of the checksum loop - while the per-iteration execution time of the loop might vary slightly from iteration to iteration, its *average* execution time over several iterations should be very stable. Each block in the main checksum loop on Arduino Uno takes about 20 machine cycles, and calculating the entire 160-bit checksum takes $\approx$ 400 cycles (about $25\mu s$). Therefore, in this paper, we use $1ms$ segment size with 80% overlap so that each segment corresponds to about 40 iterations of the checksum calculation, and consecutive segments differ only in 8 iterations of the checksum calculation.

Each SFDS then goes into the `findPeaks()` module (❹) where n spikes are selected and later used as "signatures" for each SFDS. In `findPeaks()`, the first step in finding

a spike is that for each frequency, f, that is of interest in an SFDS, we first compute the corresponding normalized frequency as $f_{norm} = (f - f_{clk})/f_{clk}$, where f_{clk} is the clock frequency for that SFDS. This normalized frequency is expressed as an offset from the clock frequency so that a shift in clock frequency does not change f_{norm} with it and is normalized to the clock, so it accounts for the clock frequency's first-order effect on execution time. We call this technique **"clock-adjustment"**. The criteria for selecting spikes are choosing the n largest amplitude local maxima (peaks), excluding the spike for the clock, that are not part of the *noise*. To find the noise, we record the spectrum once without executing the attestation function and save all the spikes that are 3-dB above the noise floor as *noise*. For our evaluations, we select $n = 6$, to capture the checksum loop's fundamental frequency and its second and third harmonics (in both sidebands).

The output of `findPeaks()` is a vector of n numbers (i.e., frequency bins). To check the correctness of the execution, we need to check this vector to a known reference model (also a list of size n) that is achieved and saved during the secure execution of `Checksum()`. Note that the reference model needs to be created only once. We assume that either the embedded system's manufacturer or the end-user is able to achieve a correct reference model. The reference model vs. SFDS comparison is a simple Euclidean distance comparison where all n peaks should be compared to the all peaks in the reference model (❺). For each comparison, if the distance is smaller than a threshold we increment a counter. Finally, if the counter is larger than $n - 1$, it means we are in the *checksum main loop* state. Based on the distance comparison, `findDistance()` outputs a boolean value showing whether we are in the checksum main loop or not.

The final stage of EM-MON is a Finite-State Machine (FSM ❻). The default state for the FSM is when EM-MON is waiting for the attestation to start ($state = 0$). Upon receiving a challenge from $\mathcal{V}$, FSM switches to $state = 1$, and starts a timer called `challengeTimer`. Once the boolean value from `findDistance()` becomes one (i.e., checksum loop starts), the FSM switches to $state = 2$ if the timer is less than a

threshold, otherwise it throws an error. Checking this value ensures that the system can be protected against *proxy*, *code compression*, and *return-oriented rootkit* attacks, where the attacker needs to spend some (non-negligible) time to set up the attack before actually starting the `Checksum()`. Note that this is an important and unique feature of EMMA since existing software-based methods [137, 138, 140, 141] are all unable to measure this delay (*even when they are directly connected to the verifier by a cable*), and can only measure the time between sending the challenge and receiving the checksum value. Thus, if the extra time overhead caused by these attacks is relatively smaller than the total attestation time, time-based methods will fail to detect these attacks.

The output of `findDistance()` becomes zero when the checksum loop completes, so the FSM switches to $state = 3$, and checks the `challengeTimer` once again. This check ensures that the total execution time of attestation does not exceed a threshold which is defined by $initTime + perIteration \times totIter$, where $perIteration$ is the checksum loop per-iteration time, $totIter$ is the total number of iterations for calculating the checksum, and $initTime$ is a constant.

Lastly, in $state = 3$, EM-MON starts a timer called `checksumTimer` and waits for an acknowledge from $\mathcal{V}$ that the checksum is received. At this point, if `checksumTimer` is larger than a constant, FSM again throws an error. Otherwise, it successfully switches back to $state = 0$ and waits for the new attestation challenge. This check ensures that the adversary can not spend any extra time after the checksum calculation is finished and before actually sending the checksum to $\mathcal{V}$. Note that in all cases, FSM can only transit from state n to $n + 1$ to enforce the correct ordering in attestation.

6.4 Evaluations

6.4.1 Measurement Setup

To evaluate the effectiveness of our method, we implemented EMMA on a widely used and popular embedded system, Arduino Uno, with an ATMEGA328p microprocessor clocked

at 16MHz. To receive EM signals, the tip of a small magnetic probe [89] was placed about 10 cm above the Arduino's microprocessor (with no amplifier). To record the signal, we used a commercially available compact software-defined radio (USRP B200 Mini-i [100]). We recorded the signals at 112 MHz (i.e., the 7th harmonic of the Arduino's clock), with a 10 MHz sampling rate. Note that all of our measurements were collected in the presence of the other sources of EM interference including an active LCD that was intentionally placed about 15 cm behind the board. A set of TCL scripts were used to control the attestation process (e.g., sending a challenge, recording a signal, etc.). The real-time EM-Monitoring algorithm was implemented in MATLAB2017b.

6.4.2 Implementation

Arduino Uno uses an ATMEGA328p microprocessor, an Atmel 8-bit AVR RISC-based architecture, with a 16KB Program memory and a separate Data memory (unlike most other architectures where a single memory is used for both data and for executable instructions). This micro-controller has 32 8-bit general purpose registers where the last 6 registers can be combined in groups of two, and form three 16-bit registers (namely X, Y, and Z). The Z register can be used to access/read the program memory using $LPM\ Z$ assembly instruction. Note that, unlike most of the micro-controller architectures, AVR does not provide direct access to the Program Counter (PC) register, so the value of the PC cannot be used during checksum calculation.

As mentioned in Section 6.3, we use a 160-bit checksum which is saved as a vector in 20 8-bit registers ($r0-r19$). Z register ($r31 : r30$) is used for reading the program memory ($memAddr$), Y register is used to store the random number generated by PRNG. Inputs from the challenge are pushed to the stack prior to invoking `attest()`, and later are read in the *initialization* phase. $r25 : r24$ are used to save the $MASK$ value. $r23 : 21$ is used to save the $nonce$, and $r20$ is used to store the content of the memory. Finally, X is used for saving the current index (i). In our framework, each partial checksum calculation ($cSUM$)

takes 20 cycles. Hence, adding an extra one-cycle instruction (e.g., an ADD) to the partial checksum block should increase the per-iteration time (and the corresponding spike in the frequency domain) by about 5%.

6.4.3 Attacks

In this part, we evaluate the security of EMMA by implementing several known attacks on software-based attestation framework and showing that EMMA can indeed detect these attacks, and protect the system against them.

1- Memory-Copy Attack:

The most straightforward attack against software attestation is *memory-copy attack*, where the adversary has created a copy of the original code *elsewhere* in memory, and the checksum code is modified to use that range of addresses instead of the original ones. Since the challenge sent by $\mathcal{V}$ could request to read any memory line in the program memory address space, potentially including the supposedly "empty" memory space where the "clean" copy of the original code is kept, to avoid detection this modified code must check addresses that are used during checksum computation, and then perform accesses without modification for unmodified memory ranges, redirect them to "clean" copies for modified memory ranges or use override values for supposedly empty ranges that now actually contain the attacker's data (including "clean" copies of original values from program memory).

This checking and redirection of memory requests introduce overheads during checksum computation. Specifically, the adversary needs to change $memAddr$ register (register Z in our implementation) to point to another address in the memory (at least one added instruction). Moreover, since we are using $memAddr$ in the checksum calculation, the value itself has to be changed back to the correct value (another one instruction). Note that, in our implementation, since accessing the program memory is only possible through Z, the adversary's only option is changing Z. Even for program/data location that are unchanged

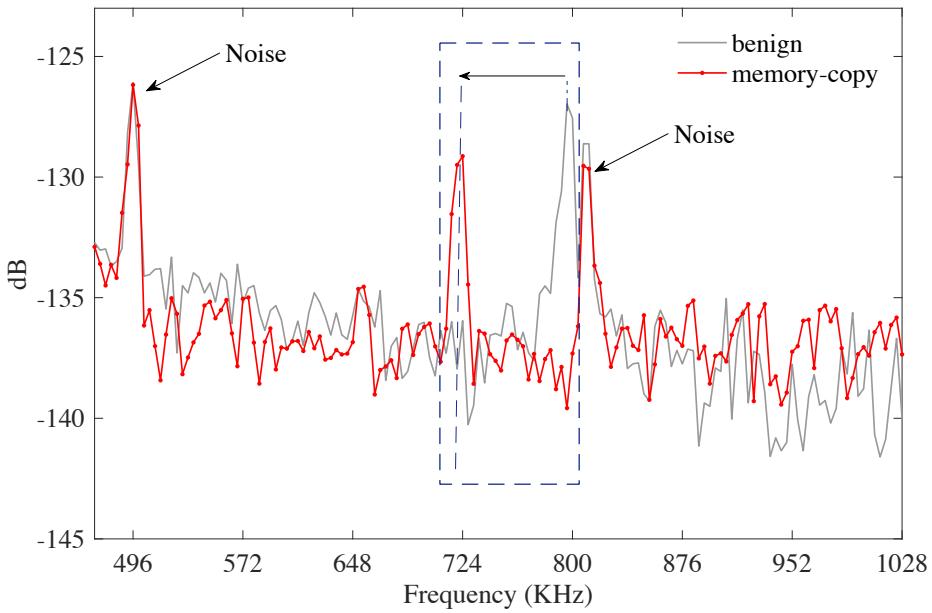


Figure 6.3: EM spectrum during checksum computation for the original code (gray) and Memory-Copy attack code (red). The x-axis is the frequency offset relative tot he processor's clock frequency.

by the adversary, the checksum code must suffer overheads of checking (a compare and a branch instruction) that the address falls in a range that still contains original instructions/data. Overall, to implement this attack, the adversary has to add at least two instructions per check-summed location.

In this work, to model a more stealthy attack, we relax the assumption that "free" space is filled with random values and allow all "free" memory locations to be filled with the same value (e.g., 0xFF). This allows the attacker to store malicious code in an empty region of program memory, and to modify checksum computation so that $LPM\,Rd,\,Z$ (i.e., load from program memory) for that region of the memory is replaced with $SERRd$ (set Rd to 0xFF). On the architecture used in our experiments, an LPM instruction uses 2 more cycles than SER does, thus removing 2 cycles from the 4-cycle performance penalty introduced by the *compare-and-branch* check, that is still needed to determine which region of program memory is being addressed. The reduced additional latency makes the modification of the checksum computation more difficult to detect.

To evaluate our framework, we implemented the Memory-Copy attack, and we trained EMMA on (only one) attack-free instance of attestation. We then applied EMMA to both attack-afflicted and attack-free instances of the attestation. The spectra of the resulting signals (Figure 6.3) show the spikes that correspond to the original checksum computation loop, and also the spikes that correspond to the modified checksum computation (red), which are shifted closer to the processor clock's frequency because the per-iteration time of the loop has increased. Figure 6.4 shows the spike's frequency for 20 attestation instances, 10 attack-afflicted, and 10 attack-free, showing a consistent difference among them. We find that EMMA successfully labels all these instances, i.e. all attack-afflicted instances are labeled by EMMA as attack-afflicted (successful detection), and all attack-free are labeled as attack-free (no false positives). The "measurement-limit" line refers to the threshold that was used in EMMA for the labeling decision.

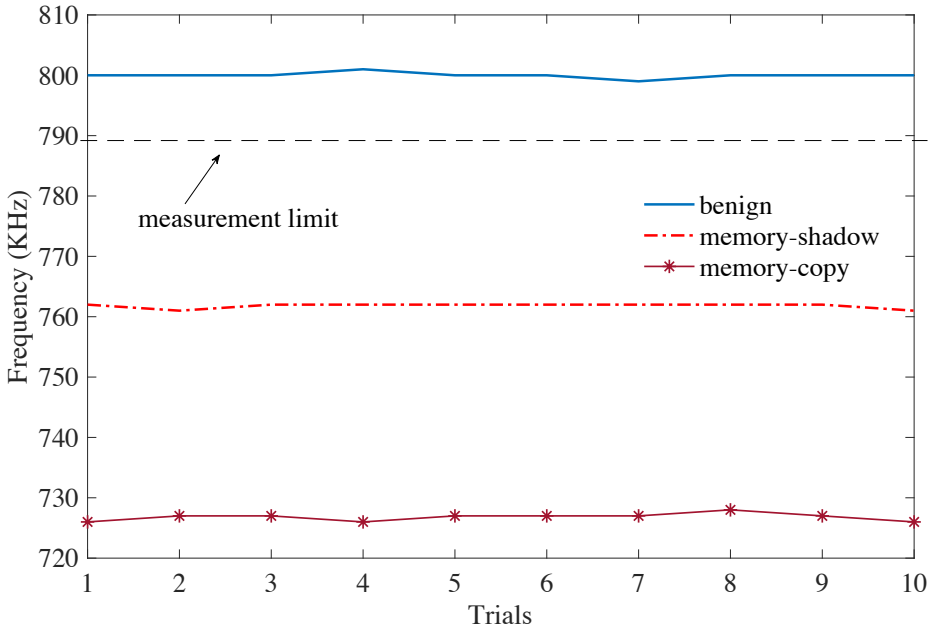


Figure 6.4: Frequency of the checksum computation loop for (a) attack-free, (b) memory-shadow attack, and (c) memory copy attack code.

2- Memory-Shadow Attack:

Castelluccia *et al.* [143] showed that, instead of adding a *branch and compare* instructions to compare the address, the adversary can reduce the number of added cycles by one (2 instead of 3) by copying the malicious code into a specific "free" area of the program memory (assuming it is not filled with random values) and called it Memory-Shadow Attack.

This attack leverages the fact that most of the program memory available on an embedded system is usually empty. In this case, if the second half of the program memory (addresses with $MSB = 1$) is empty, all accesses to the last quarter of the memory (addresses $11x...x$) could be redirected to the third quarter (addresses $10xx...x$) by just flipping one bit in the address. The last quarter of the program memory would then be shadowed and could be used to hide the malicious code. To implement this attack, the adversary simply needs to check the highest bit, and if it is one, then sets the second highest bit to zero. This can be implemented in an AVR micro-controller with two instructions "$sbrs\ r31,\ 7$" and "$cbr\ r31,\ 6$" which takes two cycles; thus, the overall number of added cycles will be three which is only one cycle more than the actual checksum loop (recall that the adversary does not need to read from the memory so that gives him an extra two cycles).

To further show that adding even an extra cycle to the main loop would significantly shift the frequency of the loop, Figure 6.5 shows the frequency shift for adding a simple one cycle ALU instruction where we added a CBR (clearing a single bit in a register) instruction to the main loop. This confirms that the adversary *cannot add any instructions* to the main loop of the code, a very strong security guarantee. Furthermore, similar to memory-copy attack, we used 10 trials for this attack and showed that its frequency is well below the actual frequency of the main loop (see Figure 6.4). Similar to memory-copy, here we also got 0% false positive and 100% true-positive rate accuracy.

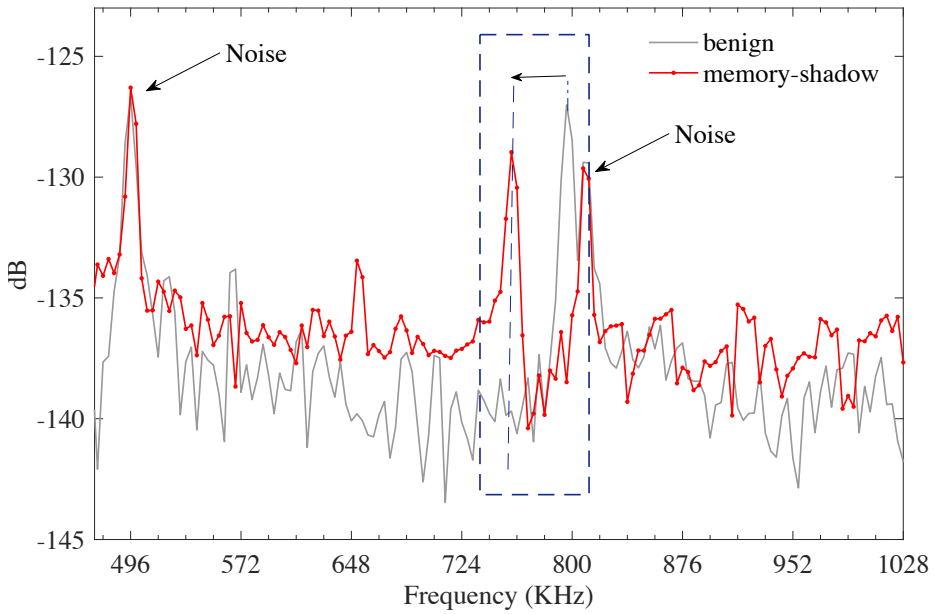


Figure 6.5: EM spectrum while executing the original checksum computation code (gray) and Memory-Shadow attack code (red).

3- Code Compression Attack:

An alternative method for hiding the malicious code is to apply compression (e.g., Canonical Huffman Encoding [146]) to the original code in the program memory, and then use the freed space to store malicious code. The compressed original code can then be decompressed *on-the-fly* as needed during the checksum computation (for each $memAddr$, decompress the corresponding block and use the decompressed value to update the checksum). However, decompression in the main loop significantly increases the loop's per-iteration time. Indeed, we have implemented the code-compression attack from [143] and found that the per-iteration time of the checksum loop increases by more than 300%. After applying EMMA to 10 instances of this attack, we found that EMMA detects all 10 instances as attack-afflicted, with no false positives in attack-free runs.

4- Rootkit-Based Attack:

Another class of attacks leaves the original checksum loop unchanged, but adds work before and/or after it, e.g., by hiding/removing the modified code before the checksum com-

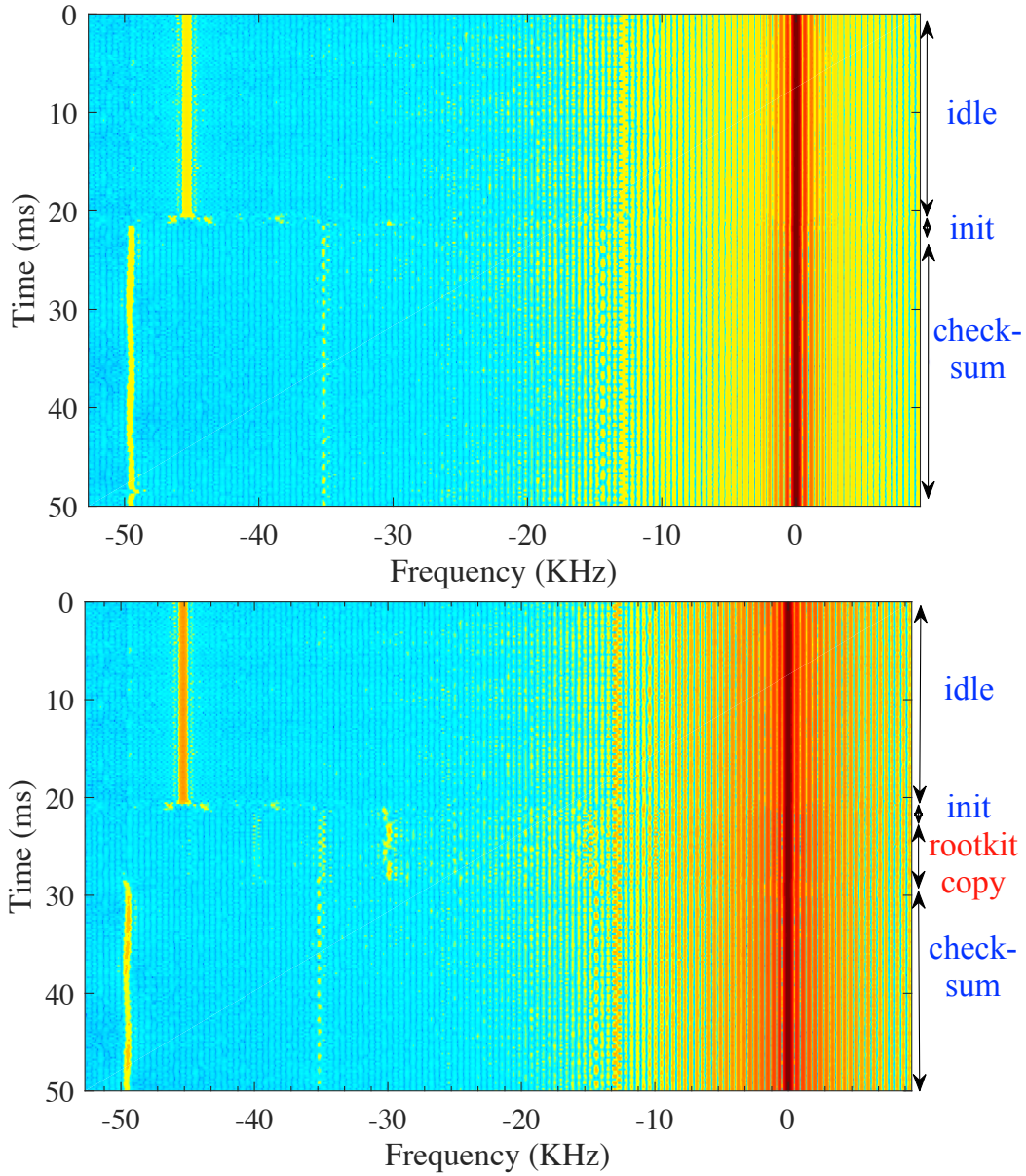


Figure 6.6: Spectrogram of the attestation code in normal (top) and rootkit-attack (bottom) runs. Note that the slight differences in colors between the two spectrograms correspond to variations in signal magnitude which are caused by different positioning of the antenna. Such variation is common in practice and has almost no effect on EMMA's functionality because EMMA was designed to be robust to such variation.

putation, and restoring it after the (unmodified) checksum computation. These types of attacks are extremely difficult to detect for existing timing-based schemes since the time overhead introduced by these attacks are typically tens of milliseconds which is less than 1% of the overall execution time of the attestation procedure. However, for EMMA, these

attacks are not difficult to detect, as they add many cycles of work between when the checksum computation is supposed to start and when it actually starts.

We implement the Rootkit-Based Attack [143] as an example, which leverages Return-Oriented Programming (ROP) technique [101]. In this attack, a hook (jump instruction) replaces the first instruction in the attestation. When attestation is initiated, this results in a jump to the malware's hiding functionality, which deletes the attacker's code (including itself) from program memory, but leaves a series of ROP gadgets so that, after the (unmodified) attestation code sends its response, the malware is re-installed on the device.

The *deleting* procedure is the most time-consuming part of the attack, where the adversary needs to copy the malware hiding functionality and the modified checksum loop to the data-memory, and replace them with the original code. Figure 6.6 shows the spectrogram of the attestation procedure with and without the rootkit-based attack. As can be seen in the figure, for the normal behavior of the attestation code, initialization takes about $1ms$ which includes receiving the challenge and invoking `attest()`. Note that based on the initialization time, we set the *threshold* for `challengeTimer` to $2ms$ or 8 samples (i.e. the maximum delay between sending the challenge and starting the checksum main loop is smaller than $2ms$). As can be seen in Figure 6.6, in the presence of rootkit attack though, there is an extra phase between the initialization and the start of the main checksum loop that takes about $8ms$, which is larger than the timer's threshold and thus triggers an error caused by checking the `challengeTimer`.

To evaluate EMMA against this attack, similar to previous attacks, we used 10 trial runs for the rootkit attack, and found that EMMA can successfully detect all the instances of the attack. i.e., our detection algorithm successfully found that there is a delay between the initialization and the beginning of the loop, and throws an error without having any false positives.

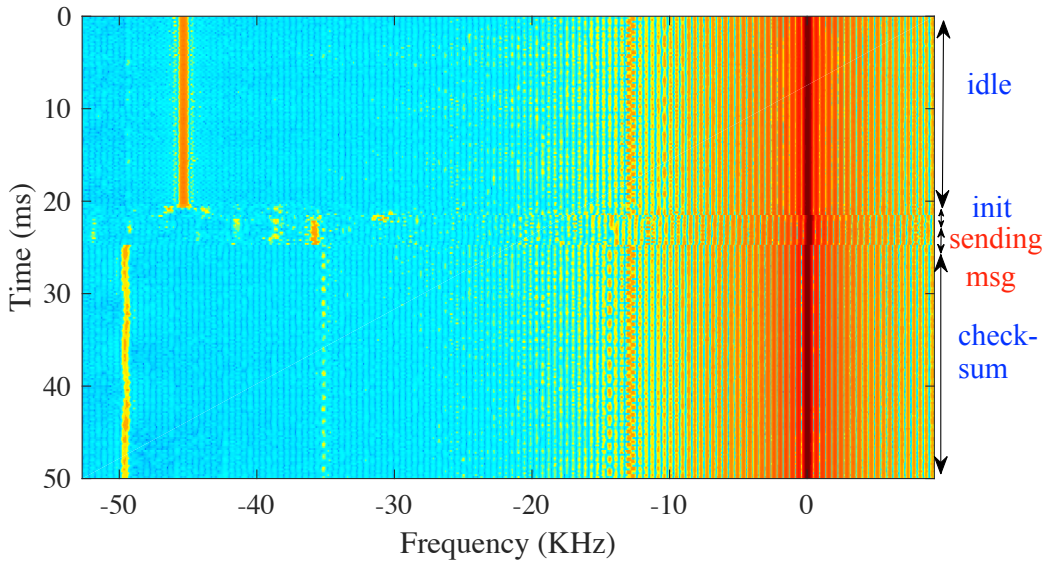


Figure 6.7: Spectrogram for the proxy attack. An extra region (sending message) can be seen in the figure.

5- Proxy Attack:

In a proxy attack, instead of calculating the checksum on its own system, the prover contacts another often faster device (the proxy) to compute the correct answer (checksum) to the time-sensitive checksum computation, which enables malware on the device to go undetected. In this case, similar to a rootkit-based attack, the adversary needs some time after receiving the challenge to properly set up the attack. For proxy, this time is used for sending (forwarding) the challenge and any other necessary information (e.g., nonce) required to correctly compute the checksum in the proxy.

The major limitation in the existing software-based attestation methods for detecting proxy attacks is that the adversary can simply hide this attack if $t_{send} << t_{threshold}$. EMMA, however, is not limited by the overall attestation time and can distinguish the initialization phase from checksum calculation very accurately.

Figure 6.7 shows the spectrogram of the attestation procedure with and without the proxy attack. In our proxy attack, we sent the challenge through the serial link back to the PC to imitate sending message operation of the proxy attack. As can be seen in the

figure, sending the challenge added approximately extra $4ms$ to the initialization phase, and has a spectral signature that is completely different from that of in the initialization or the checksum calculation phases.

To evaluate EMMA against this attack, we used 10 trial runs for the proxy attack, and found that EMMA can successfully detect all the instances of the attack. i.e., EMMA successfully found that there is a delay between the initialization and the beginning of the loop, and throws an error while having no false positives.

6.5 Further Analysis on Scalability and Robustness

6.5.1 Scalability to Other Platforms

To show EMMA is applicable to other embedded systems with a different processor and/or architecture, and even can be used at different frequency ranges, we tested our checksum main loop, `Checksum()` (an un-optimized form), on three other embedded systems including a TI MSP430 Launchpad with a processor clocked at 16MHz, an STM32 ARM Cortex-M Nucleo Board also clocked at 16MHz, and an Intel Altera's Nios-II soft-core implemented on a Terrasic DE0-CV FPGA development board clocked at 50MHz. The criteria for choosing these boards were to pick the embedded systems that are popular and widely used, and have different architectures than Arduino's processor (i.e., AVR).

Running the same attestation code on these boards, we then confirmed that by using the same setup used in the previous section, EM-MON receives EM signals similar to that of for Arduino board (i.e., spikes at the frequency of the loop), and further, we confirmed that our detection algorithm can successfully detect when this loop starts and when it ends by adding the training information (i.e., the position and the number of spikes) for each board to our framework. Finally, to further show that even single added instruction to the loop's code is detectable by EMMA, we added a single-cycle "ADD" instruction to the checksum's assembly code for each of the mentioned boards (i.e., similar to *memory-shadow attack*). Figure 6.8 shows the spectrum for the checksum loop once with this extra

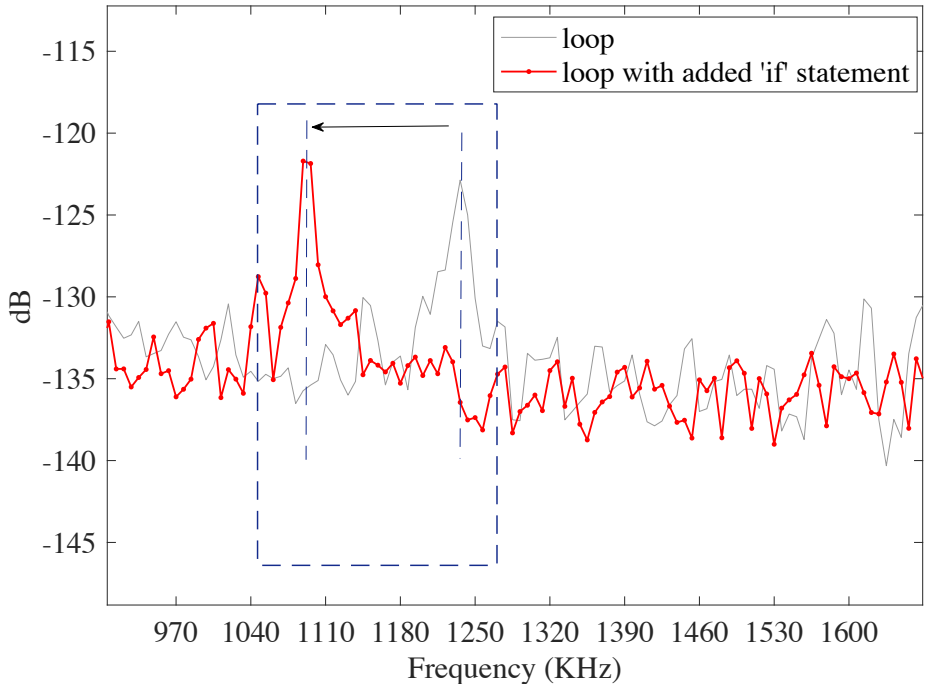


Figure 6.8: The spectrum for the Nios-II processor with (red) and without (gray) adding malicious "ADD" instruction.

added instruction and once without it. As can be seen, adding an extra instruction has shifted the frequency by about 100KHz. We then used EMMA, to label malware-free and malware-afflicted (i.e., runs with the extra instruction) checksum runs (10 each) for all the three boards. Our results showed that, in all cases, EMMA successfully detected the malicious runs.

6.5.2 Scalability to More Complex Systems

We tested our framework on a more complex system, A13-OLinuXino Single-Board-Computer. This board has an ARM A8 in-order core clocked at 1GHz, with 2 level of caches, a branch predictor, and a prefetcher. It also runs a Debian Linux OS. We ran our checksum loop on this board and measured the beginning/end time and the loop's per-iteration time.

Our measurements showed that while having caches, a branch predictor, and a prefetcher introduces some variation in the per-iteration time of the loop, in practice this variation is not significant. For the cache, since the checksum code is small, it fits completely inside the

CPU's L1 instruction cache. Furthermore, the memory region containing the verification function is small enough to fit inside the CPU's L1 data cache. Thus, once the CPU caches are warmed up, no more cache misses occur. The time taken to warm up the CPU caches is a very small fraction of the total execution time. As a result, the variance in the execution time caused by cache misses during the cache warm-up period is negligible.

For the branch predictor, we observed that in our code, the branch mis-prediction only happens in the last iteration (recall that the inner loop was enrolled), when the checksum computation is finished; thus, it doesn't have any impact on the execution time of the checksum loop. Furthermore, due to the memory's random access pattern in the checksum loop, the prefetcher's accuracy is inevitably low.

To further analyze the effect of having cache, branch predictor, and prefetcher on the timing, we used gem5 [147] simulator, to simulate the checksum code on an in-order ARM core machine with similar configurations to that of in A13-OLinuXino board. Our simulation results showed that for a 1.2KB size checksum code, our code accessed the cache about 7000 times, out of which only 21 accesses were L1 miss (i.e., $> 99.5\%$ hit-rate), and only about 800 more cycles (mostly due to L2 misses) were added to the overall execution time (i.e., $< 0.01\%$). Note that this extra delay only happens inside the checksum loop, and does not affect the delay for the proxy and/or rootkit attacks since those attacks happen *before* the beginning of the checksum. Furthermore, our results showed no mis-prediction for the branch predictor, and $< 1\%$ prefetching accuracy for the checksum loop.

To evaluate EMMA on this board, we ran the same experiment (adding an extra "ADD" instruction) discussed in the previous section, and found that our detection algorithm successfully detected all instances of the attack with no false positive.

Overall, the goal of evaluating EMMA on multiple different boards was showing that the ability to use EM signals for monitoring the attestation procedure is a result of a fundamental connection between repetitive program behavior and the spectra of resulting side-channel signals, and is not dependent to a specific architecture. Furthermore, these experi-

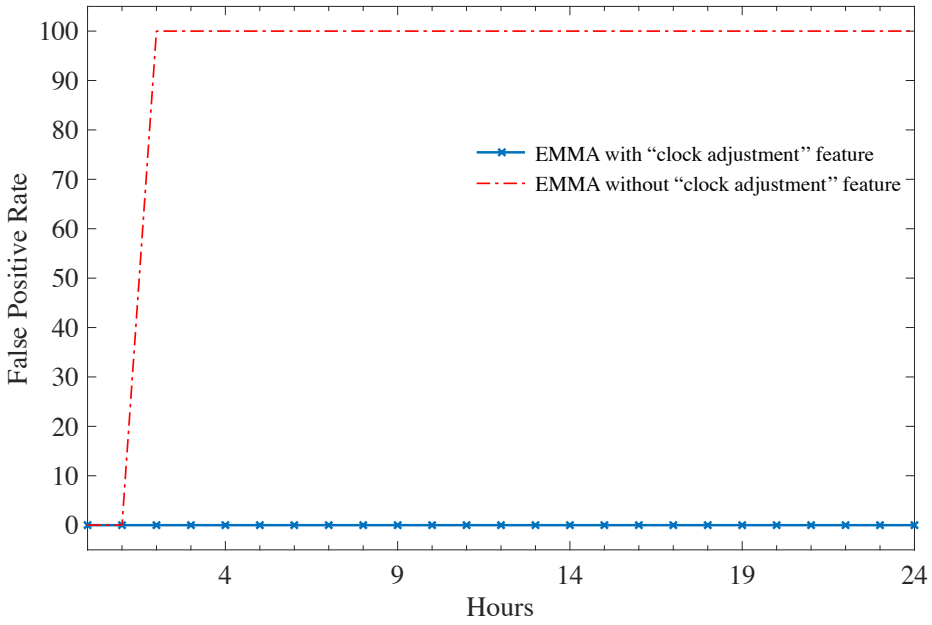


Figure 6.9: EMMA false positive rate (lower is better) with (solid blue) clock-adjusted feature and without it (dashed red) over 24 hours interval.

ments confirmed that EMMA is scalable to different and more complex architectures.

6.5.3 Robustness and Variations Over Time

To test the robustness of our algorithm over time and against environment variations (e.g., temperature, external radio interference, etc.), we repeat the attestation procedure at one-hour intervals, over a period of 24 hours, while keeping the Arduino board and the receiver active throughout the experiment, to observe how the emanated signals vary over time as device temperature (and room temperature) and external radio interference such as WiFi and cellular signals change during the day and due to the day/night transition. At each hour we ran the attestation once (without any malicious behavior). The training data was collected before the first hour of the experiment. The goal of the experiment was to show how the false positive rate changes over time. We observed a significant increase in false positive rate after hour 2 (see Figure 6.9) when we were not using the clock-adjustment feature (see Section 6.3). However, adding this feature, EMMA achieved perfect accuracy (i.e., 0% false positive). The major reason for this dramatic degradation in accuracy was due

to the fact that the clock rate for Arduino began to drift after about one hour of continuous usage. Without associating this drift to our detection algorithm, EMMA was unable to correctly predict that the shift in the frequency of the checksum main loop was due to the clock drift, and not because of a potential malicious activity.

6.6 Conclusions

This chapter proposed a new approach for hardware-software attestation on embedded systems, and described and evaluated EMMA, a proof-of-concept implementation of this approach. Unlike prior attestation schemes, EMMA uses EM side-channel signals emanated by the embedded system during response computation, to confirm that the device has, upon receiving the challenge, actually computed the response using the valid program code for that computation. This new approach requires physical proximity, but imposes no overhead to the system, and provides accurate monitoring *during* the attestation. We implemented EMMA on a popular embedded system, Arduino UNO, and evaluated our system with a wide range of attacks on attestation integrity. Our results showed that EMMA can successfully detect these attacks with high accuracy, and that it outperforms existing systems in terms of security guarantees, scalability, and robustness. Further, we showed that EMMA can be scaled to attest multiple embedded devices, that is can support other embedded systems/platforms, and that it is robust against various sources of variability.

We envision that EMMA can be used to attest a group of embedded systems that are mostly dedicated to a specific task. This includes, but is not limited to, a network of sensors or peripherals that are connected to a main controlling unit, cyber-physical systems in hospitals and/or factories, industrial IoT (IIoT) systems, etc. In these scenarios, the cost (per device) and complexity of deploying EMMA is relatively low because it requires no changes to the monitored device, and thus creates no regulatory, safety, or disruption concerns for the system. More importantly, it has zero-overhead on the monitored system and is physically separated from the monitored device. In a practical scenario, EMMA can

leverage an already existing infrastructure for controlling the CPS such as industrial control systems (ICS), SCADA, etc. which further simplifies its implementation and reduces the costs. Furthermore, we envision EMMA is being useful in other scenarios such as checking the integrity of legacy systems (which are notoriously difficult to manage and verify), providing a secure execution environment on sensor nodes and/or IoT devices for secure code update, error recovery, key exchange, etc. Finally, EMMA can be used as a *portable* setup to occasionally monitor one or a small group of devices. In this scenario, EMMA can be used as a low-cost, powerful tool to debug under-the-test systems.

CHAPTER 7
A NEW SIDE-CHANNEL VULNERABILITY ON MODERN COMPUTERS BY EXPLOITING ELECTROMAGNETIC EMANATIONS FROM THE POWER MANAGEMENT UNIT

7.1 Abstract

This Chapter presents a new *micro-architectural* vulnerability, which is created by power management units of modern computers and can be exploited through electromagnetic, and potentially other, side-channels. The key observations that enable us to discover this side-channel are: 1) in an effort to manage and minimize power consumption, modern microprocessors have a number of possible operating modes (*power states*), in which various sub-systems of the processor are powered down, 2) for some of the transitions between power states, the processor also changes the operating mode of the voltage regulator module (VRM) that supplies power to the affected sub-system, and 3) the EM emanations from the VRM are heavily dependent on its operating mode. As a result, these state-dependent EM emanations create a side-channel that can reveal which programs are currently executing, and potentially other sensitive information about the executed programs.

To demonstrate the feasibility of exploiting this vulnerability, we create a *covert channel* that uses changes in the processor's power states to *exfiltrate* sensitive information from a system that is otherwise secured and completely isolated (*air-gapped*), and then receives that information using a compact, inexpensive receiver placed in proximity to the system. To demonstrate the *severity* of this vulnerability, we also show that information can be successfully exfiltrated even if the receiver is *several meters* away from the system, and even if the system and the receiver are separated by a wall. Compared to the state-of-the-art, the proposed covert channel has >3x higher bit-rate. Finally, to demonstrate that this new

vulnerability is not limited to being used as a covert channel, we demonstrate how it can be used for attacks such as *keystroke logging*.

Compared to prior attacks on PMUs [148, 149, 29], which could be carried out *remotely*, the proposed side-channel has a different attack model - prior attacks are practical mostly for cloud servers where unrelated (potentially hostile) services/VMs end up being co-located on the same physical server, while the vulnerability we present is mostly applicable to mobile devices, computers used in offices adjacent to public spaces, etc. Given these differences, our findings are important since the discovered side-channel is applicable even when a computer is highly secured from untrusted users, e.g., when it is physically isolated (air-gapped) from other networks. Moreover, as we will show in this paper, compared to the existing physical side/covert channel attacks [29, 41, 55, 149, 150, 151, 152] that can successfully attack an isolated system, the discovered side-channel can exfiltrate data with much higher data-rate, and in many cases, it can be received from much further distances.

This work makes the following contributions:

- Describes a new physical side-channel vulnerability that exploits the signals (e.g., EM emanations) produced by the system's voltage regulator module to infer the processor's *power-states*,
- A proof-of-concept exploitation of this vulnerability by creating a covert channel with low bit-error-rate, and with a high data-rate to exfiltrate data from an air-gapped computer,
- A practical demonstration of data exfiltration at a distance and through a wall in an office environment.
- A proof-of-concept design and implementation of a keystroke logging framework by exploiting the proposed side-channel.

7.2 Background

7.2.1 Power Management in Modern Systems

To improve energy efficiency, modern computer systems, especially mobile ones where energy efficiency directly affects battery life, employ a number of power-management techniques. One of the most popular such techniques is dynamic voltage-frequency scaling (DVFS) [153, 154], where the processor's clock frequency can be adjusted dynamically depending on the level of performance that is required. The reduced clock frequency reduces power consumption by executing fewer instructions (and thus spending less energy) per unit time. Furthermore, the speed at which the processor's circuitry can operate is dependent on its operating voltage, so the processor's operating voltage can be lowered as its operating frequency is reduced, which (dramatically) reduces the energy consumed per instruction executed. Another very popular technique consists of placing unused units within the processor into a low-power state, typically by no longer clocking the unit (clock gating [155]), but in some cases also by further reducing the unit's voltage level or even completely powering it down. Most modern processors deploy both sets of techniques. For example, the Demand Based Switching (DBS) [156] technology in Intel's processors provides the processor with a number of *performance states* (P-states), each with a different voltage-frequency value, and also a number of *processor states* (C-states) which correspond to different levels of low-power idleness. Recent processors can have more than 10 different P-states, where P0 is the highest-performance state, and higher state numbers correspond to lower voltage-frequency settings (and thus lower performance). For Intel processors up to Haswell/Broadwell architecture, the desired P-state is specified by the operating system, by writing the corresponding value into a special processor register, and the processor's hardware simply implements the specified voltage-frequency settings [157]. More recently (starting with the Skylake architecture [158]), the operating system can leave the control of the P-states to the processor's hardware (called Speed-Shift technology [156] by Intel),

and this is the default behavior in recent operating system releases because it enables more rapid P-state adjustments as the processor's workload changes.

Whereas P-states allow management of the tradeoff between the processor's performance and energy consumption while active, the *C-states* are a set of *low-power* modes that the processor can switch to when it is idle. The C0 state corresponds to the processor's normal operation (execution of instructions), whereas C1, C2, etc. states correspond to idleness with increasing levels of clock- and power-gating for the units within the processor. Thus higher-numbered C-states save more energy while the processor is idle, but also take more time to "wake up" the processor. Typically, states C1 through C3 only apply clock-gating, C4 through C6 reduce the voltage, and new Enhanced C-states can do both at the same time. The transition between C-states relies on a set of sensors that monitor utilization of the cores, but the actual algorithm for choosing when and which C-state to use is not publicly available (and may change from one generation of processors to the next).

The P- and C-states are enabled by default but, on all recent laptops we examined, the BIOS has settings for disabling them (at significant cost in terms of power-efficiency). Further, P-states can also be controlled through the OS using tools provided by the kernel (e.g., *cpufrequtils* in Ubuntu).

7.2.2 Voltage Regulator Module (VRM)

VRM supplies power to the cores. The processor uses a set of Voltage Identification (VID) hardware signals to inform the VRM which voltage-level to provide [159]. The VRM is typically an integrated circuit that is soldered onto the system's motherboard, but in some recent processors (e.g., Intel's Haswell architecture) the voltage regulator is integrated into the processor's package (an Integrated Voltage Regulator, or IVR), or even into the processor's silicon die (a Fully Integrated Voltage Regulator, or FIVR).

In laptops (and desktops, too), the most commonly used style of a voltage regulator is a *Buck* [160] converter (also called step-down converter). It is a DC-to-DC power converter

which is connected at its input to a higher-voltage DC supply (e.g., the laptop's battery pack or AC-power adapter, which typically supply 10-20 V), and outputs a lower voltage, typically between 0.7 V to 1.4 V to its load.

Intuitively, a Buck converter consists of a capacitor at its output that it tries to keep filled to the desired voltage level. Since the load draws current from this capacitor, the charge it holds drains over time, causing its voltage to drop. To compensate for this drop, the converter periodically connects the capacitor to its input, causing a burst of current that replenishes the capacitor's charge, thus bringing the output voltage back to the desired level. The amount of time between these replenishments (***switching period***) in computer-system VRMs is typically a few (1-4) microseconds.

The VRM must be capable of supplying enough current under maximum-load conditions, so its switching period must be short enough that even the maximum output current does not drain the output capacitor below the minimum required level. At low load currents (e.g., when the processor core is idle), however, the voltage regulator becomes less efficient – it switches just as often as under full load, thus wasting a similar amount of power on switching losses, while the power actually provided to the load is very small, so the switching losses become a much larger fraction of the overall power consumption. Since the low load current also means that the VRM's output capacitor is still almost fully charged at the end of each switching period, a typical VRM improves its low-load efficiency using a technique called ***phase shedding*** [161, 162, 163], where for some of the switching periods the VRM does not switch, skipping the replenishment of the still-almost-full capacitor and saving the energy that would have been wasted to do so.

7.3 Side-Channel Vulnerability on the Power Management States

To demonstrate that switching between *active* and *idle* power states (i.e., P-states and C-states) creates a distinguishable signal, we perform a simple experiment where the system is alternating between an idle and an active state, and the received EM signal is analyzed

```
1  void microbenchmark(int t1, int t2){
2    int dummy;
3    while (1) {
4      // active state
5      for(int i=0; i<t1; i++)
6        dummy += dummy + i;
7      // idle state
8      usleep(t2);
9    }
10 }
```

Figure 7.1: The micro-benchmark used in this paper to generate ACTIVE and IDLE states for the processor to create EM side-channel signals.

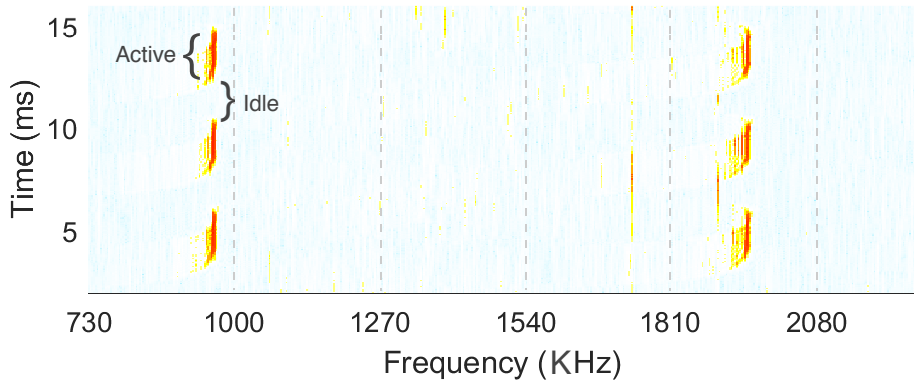


Figure 7.2: Alternation between *active/idle* states and the spikes (and its first harmonic) created by the emanated EM signals from PMU shown in the frequency domain over time.

to find whether this alternation can also be found in the signal. As described in section 7.2, due to the way VRM operates, during the *active* state, we expect to observe strong (in terms of magnitude) spikes, and weak spikes during the *idle* states. We use a program shown in Figure 7.1 to create such alternations. This program creates an infinite loop that continuously performs some activity (e.g., addition) for a while followed by a period of idleness. The duration of the active period is controlled by the value of t_1, while the duration of idleness periods is controlled by the value of t_2. Note that the actual activity (lines 5-6) can be any processor-intensive activity. Similarly, the idle periods can be implemented in any way that leads the OS to believe that the processor will be inactive for a while.

The resulting EM signals emanated from the PMU is depicted in Figure 7.2. This figure shows how the measured signals change over time in the frequency domain (i.e. a

spectrogram), where the higher intensity illustrates higher signal amplitude. A repeated pattern of strong/weak spikes is present, as expected for this side-channel. The frequency of the spikes also matches with the expected frequency of the PMU (i.e., around 970KHz) for the tested laptop[1]. To further examine the received signals, we changed the lengths of *active* and *idle* periods by changing t_1 and t_2, and observed that the length of the spikes (i.e., the red lines shown in Figure 7.2) *do* change as periods change.

To further confirm that these spikes are indeed related to the VRM and caused by the changes in the processor's power states (i.e., P- and C-states in Intel's processors), we repeat experiment with disabling/re-enabling of P-states and C-states in the system's BIOS (i.e., changing DVFS settings) and examine how that affects the received EM signals and their spectra. We found that, even when either C-states or P-states (but not both) are disabled, we observe a signal spectrum similar to that in Figure 7.2, i.e., the spikes in the spectrum appear and disappear (with no change in the data transmission rate). However, when both C-states and P-states are disabled, the spikes in the spectrum have a much stronger magnitude but are continuously present regardless of the program activity. This is consistent with our expectations - when the processor's management of power states is disabled, the processor is forced to operate at its nominal operating voltage and frequency regardless of its workload, even when the system is "idle" [2]. This forces the VRM to continuously remain in its high-power mode. The results of experiments where only C-states or only P-states were disabled are also consistent with our expectations - in those cases, the processor can still switch between idle and active states (e.g., C_0 and C_N for C-states, or P_N and P_M where $M > N$ for P-states). This observation indicates that, fundamentally, to observe this side-channel, the processor needs to be able to switch between at least one high-power and at least one low-power state (which can be different C-states, different P-states, or a combination of both).

[1]These emanations are around the clock frequency of *PMU* and not processor's clock frequency since it is created by PMU and not the CPU.

[2]When the system is idle while the C-states are disabled, it actually runs the operating system's "idle" process, usually an infinite loop, so the system's processor is not actually idle.

Attack Model. Based on the observations above, an attacker, $\mathscr{A}$, can exploit these signals in two meaningful ways: ***(i)*** The attacker can create a covert channel by intentionally forcing the processor to alternate between periods of high activity and periods of idleness, according to the values of (secret) data bits she desires to exfiltrate. Section 7.4 describes, in detail, how such a covert channel can be created. ***(ii)*** The attacker can monitor the emanated signals to infer *(a)* whether the processor has become active or not. Such information can be particularly helpful to find, for example, whether a key is pressed. Further, the attacker can monitor these signals to infer *(b)* how long the processor was active. Such information, for example, can be used for website fingerprinting (i.e., by measuring how long it takes to load a webpage, the attacker can infer which website was loaded). Section 7.5 describes how this side-channel can be leveraged for keylogging in details.

7.4 Covert Channel Communication

7.4.1 Transmitter Design

To create a covert channel, the *transmitter* (also called source or data sender) application, which has access to the secret data, should create the side-channel signal depending on this sensitive information. The transmitter code is shown in Figure 7.3. For each bit of data, depending on the value of the bit, the code either performs some activity for a while followed by a period of idleness (i.e., *return-to-zero* encoding [19]) or only a (longer) period of idleness. In this code, the duration of the active and idleness periods are controlled by the value of `LOOP_PERIOD` and `SLEEP_PERIOD` respectively.

Note that none of this code requires elevated (e.g., root-level) privileges, i.e., ***in our threat model***, the attacker's program runs as a user-level process that (i) has access to the data it desires to exfiltrate, but (ii) is denied access to any I/O that would allow it to send that data out of the system. Given the simplicity and brevity of the code, any number of programming languages can be used for this purpose, including most scripting languages

```
void transmitter(){
 char bit; int dummy1;
 FILE *file = fopen("secret", "r");
 while ((bit = getc(file)) != EOF) {
   if(bit == '1') {
      for(int i=0; i<LOOP_PERIOD; i++)
        dummy1 += dummy1 + i;
        // keeping the processor active
        usleep(SLEEP_PERIOD);}
        // return-to-zero coding
   else usleep(SLEEP_PERIOD * 2);
 }
 fclose(file);
}
```

Figure 7.3: The "transmitter" code for the covert channel communication.

or even shell scripts. Also, note that methods for creating such a malicious code inside an air-gap computer are abundantly represented in the research literature (e.g., advanced persistent threat [164]), although we do not discuss them in this paper.

The `LOOP_PERIOD` and `SLEEP_PERIOD` parameters in this code determine the bit-rate of the channel - in general, smaller values result in higher data rates. However, in practice, the bit-rate is limited by several practical constraints. First, the active period also includes the execution of the library and system code that implements the actual call to `usleep` and its house-keeping activity, so even when `LOOP_PERIOD` is zero the actual active period includes execution of tens or hundreds of instructions. The usable values of the `SLEEP_PERIOD` are also limited, mostly due to the granularity and precision of time measurement for `usleep` and the variability in the time needed to exit the idle state. Finally, the duration of the active and the idle phase should be roughly similar. Based on our experiments, we found that around 10μs is the limit below which the actual idleness period of $usleep()$ becomes highly variable[3].

[3]Even the manual page for $usleep()$ states that the sleep time may be lengthened slightly due to other system activities and hence cause some randomness to the overall timing.

7.4.2 Receiver Design

1) Signal Acquisition

In conventional communication systems, the transmitter, the receiver, and the signal transmitted between them, are all carefully engineered to achieve a low bit-error-rate (BER) while sustaining high throughput. This includes maintaining good synchronization between the transmitter and receiver to minimize insertion and deletion of bits, a sophisticated encoding of data bits into the amplitude and phase of the transmitted signal, etc. Furthermore, the carrier frequency at the transmitter is chosen carefully to support the desired range of distances between the transmitter and receiver. However, covert channels rely on signals that are produced unintentionally, thus, there is no control over the transmitter design, i.e., the carrier frequency and its stability, the range in which the signal's amplitude and phase can be changed depending on the activity, how stable the duration of these changes is, etc. Therefore, after the signal is received, the detection algorithm must deal with the problems of discovering when each transmitted bit begins, changes in the signal's amplitude, etc.

As discussed in Section 7.3, the observed signal patterns when the transmitter code is executing is similar to Figure 7.2. The key observation from this figure is that the received signal in frequency domain behaves like on-off keying (OOK) for the considered frequency components. Therefore, signal power level for each bit will be enough to identify the received bit. Leveraging the knowledge that there exist many frequency components related to the transmitter activity, we acquire the signal as

$$\mathbf{Y}[n] = \sum_{k \in \mathscr{S}} \texttt{abs}\left(\mathscr{F}_n[k]\right), \tag{7.1}$$

where $\mathbf{Y}[n]$ is the signal of interest, $\texttt{abs}(\bullet)$ takes the absolute value of its argument, $\mathscr{S}$ is

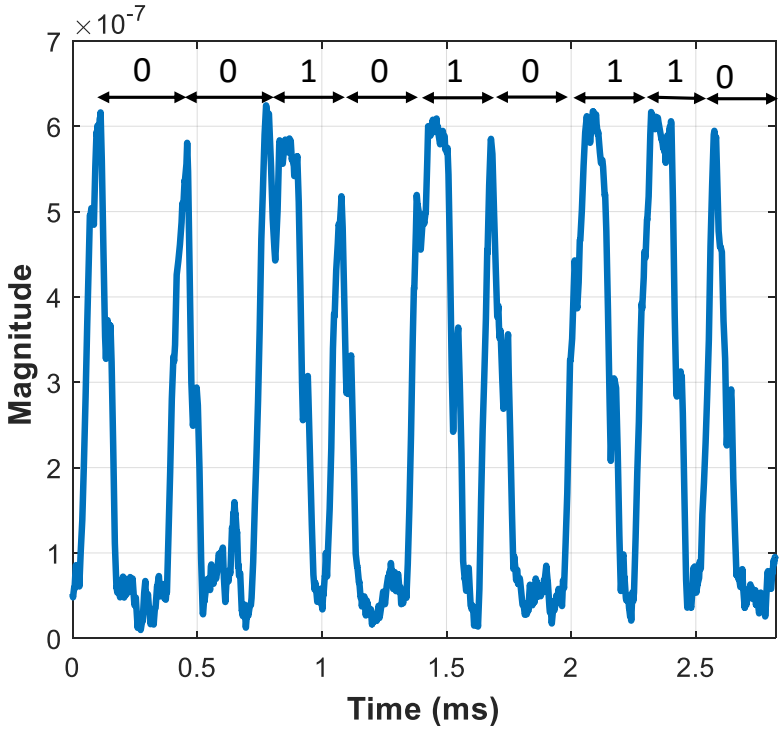


Figure 7.4: Average magnitude of the considered frequency components and the corresponding bit sequence.

the set of considered frequency components,

$$\mathscr{F}_n[k] = \sum_{m=0}^{M-1} r[m - M + 1 + n] e^{-2i\pi km/M},$$

M is the size of the Fast Fourier Transform (FFT) function and $r[m]$ is m^{th} sample of the received signal. The goal here is to increase the difference in magnitude between bit 0 and bit 1 to minimize the error-rate of the covert communication.

An example for $\mathbf{Y}[n]$ is given in Figure 7.4 where we only used the fundamental frequency and its first harmonic from the actual signal. We also plot the time-interval for each signal (called *signal timing*) and the transmitted bit. The main observations from this figure are the following:

- The signal exhibits a sharp increase whenever a new bit is transmitted, even when the bit is a zero, because processor activity is needed to execute the program code that cleans

up at the end of a previous `usleep`, reads a new bit of data, and begins a new interval.

- The magnitudes are affected by not only the additive noise but also by the variations in the execution of the transmitter.
- Due to these variations (especially the sleep time), the duration of a signal that corresponds to one "transmitted" bit varies among instances of these bits, even when the bits have the same value.

It is a common practice for the conventional communication systems to use a *matched filter* and sample the filtered signal at each symbol (bit), but that approach assumes that the symbols have practically no variation in their duration, i.e., the transitions from symbol to symbol are synchronous with a highly stable clock, and can thus be re-created at the receiver accurately. We found that, when applying the matched filter approach to our received signal, the BER was high, and upon further investigation we found that the main reason for this is the asynchronous nature of the signal - the actual bit positions in the signal quickly become misaligned with the clock created by the receiver in an attempt to match the transmitter's symbol-rate. Therefore, we had to devise a more robust (but also more computationally intensive) method for determining the timing of each bit.

2) Signal Timing for the Covert Communication

We determine signal timing (i.e., the time interval for each bit) using *batch processing*, i.e., we determine the timing of the bit by examining not only the signal that corresponds to that bit, but, also by considering a number of bit periods that precede and follow it which, in turn, reduce error-rate significantly while adding negligible detection latency.

The first step for batch processing is to find the starting locations of each bit by knowing that the derivative on these edges is almost infinity. To mimic the derivative operation, we convolve the batch signal with a vector of length, l_d (which depends on the sampling-rate). Half of the vector is set to one and the rest is set to minus one. This convolution is followed by finding the local maximum points of the convolution. An example of the process is

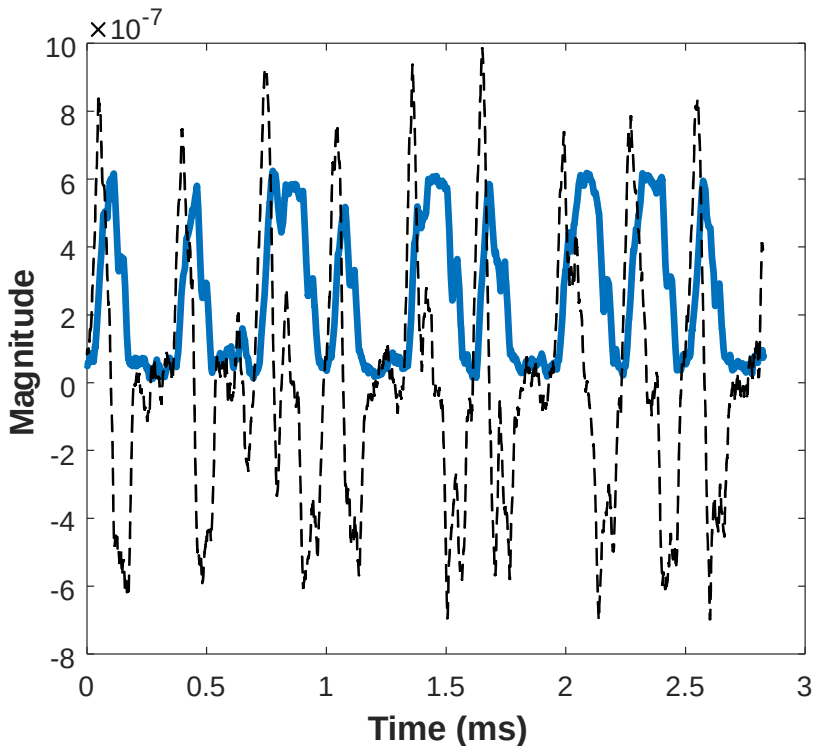


Figure 7.5: Obtaining the start points of each bit by exploiting that sharp increase whenever a bit is transmitted.

shown in Figure 7.5 where the black dotted line shows the result of the convolution operation. As seen from the figure, the resulting points of the convolution peak at the edges of $\mathbf{Y}[n]$ which indicate the starting points of a bit transmission.

Obtaining the starting points of the transmitted bits helps to find the expected signaling time of each bit. Next, we calculate the distances between the starting points of subsequent bits. The probability density function (PDF) of the distances between subsequent bits are given in Figure 7.6. This figure illustrates that the signal time has a Rayleigh distribution. The tails of the distribution indicate that some of the bit locations could not be captured because the distances between points have a positive-skewed distribution (which results in detection errors that will be discussed later).

Having the signaling time of the transmitted bits helps to fill the gaps that the detection algorithm could not find at its first attempt. We choose the signaling time of the covert communication as the point whose cumulative probability distribution equals to 0.5 since

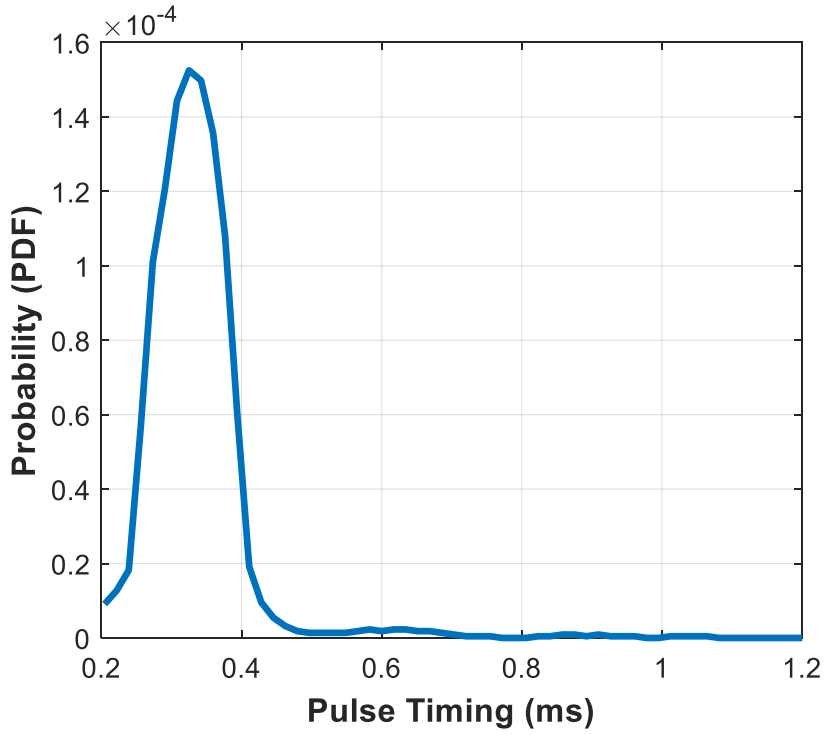


Figure 7.6: Pulse width variation of the covert communication system.

the distance distributions accumulate around this point and using the median value can minimize the false insertion and deletion rate of bits.

3) Signal Labeling Based on Average Signal Power

The variation in signal timing can also cause incorrect labeling of the received bits. The total power of the received signal could be high only because the usually very short active part of the signaling period has lasted much longer than usual. Therefore, while decoding the received signal, the algorithm also needs to take into account the variation of the signaling period's duration.

Specifically, the receiver detection algorithm utilizes the average power of the received signal samples for each bit. Let's assume we have the samples $s[n] \in \{0,\ 1,\ \cdots,\ N-1\}$

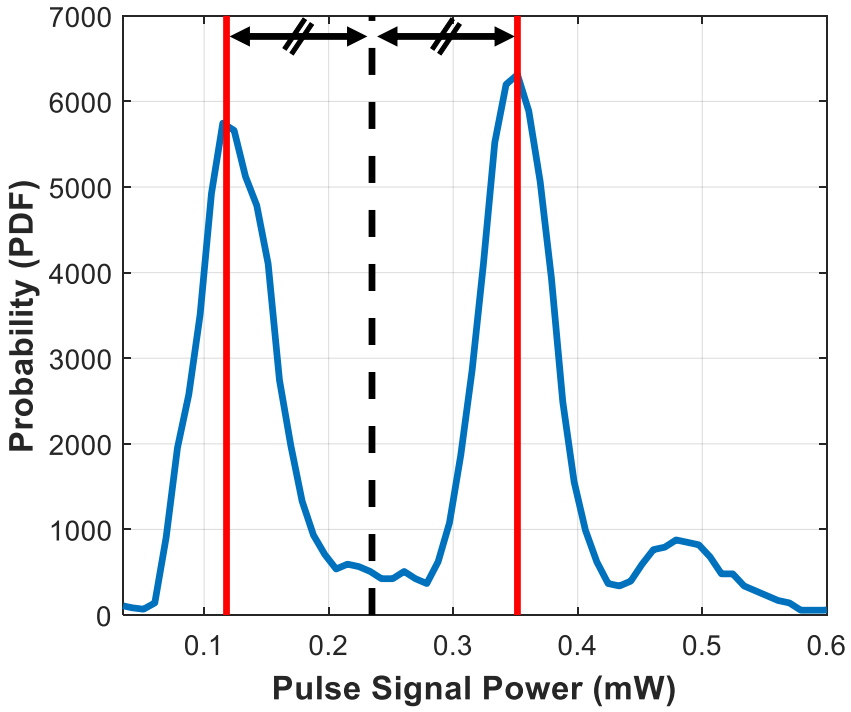


Figure 7.7: The power distribution of the pulses generated by the power management unit for IDLE (left) and ACTIVE (right) states.

for the m^{th} received bit. The detection algorithm label the received bit as one if

$$\frac{1}{N}\sum_{n=0}^{N-1}|s[n]|^2 > \texttt{thr}, \tag{7.2}$$

where `thr` is the threshold value. However, the threshold value must be chosen carefully to minimize the error-rate. Figure 7.7 illustrates the distribution for the average signal magnitude for each bit. We observe that there exist two peaks which indicate the power of bit zero and bit one. Therefore, the algorithm selects the threshold as the mean of the points corresponding to these two peaks. This threshold selection process is also illustrated in Figure 7.7. Red lines in the figure correspond to the local maximum of average power distribution, and the dotted black line is the selected threshold value for the batch. The equal signs on the arrows mean that the distances indicated by these arrows are equal.

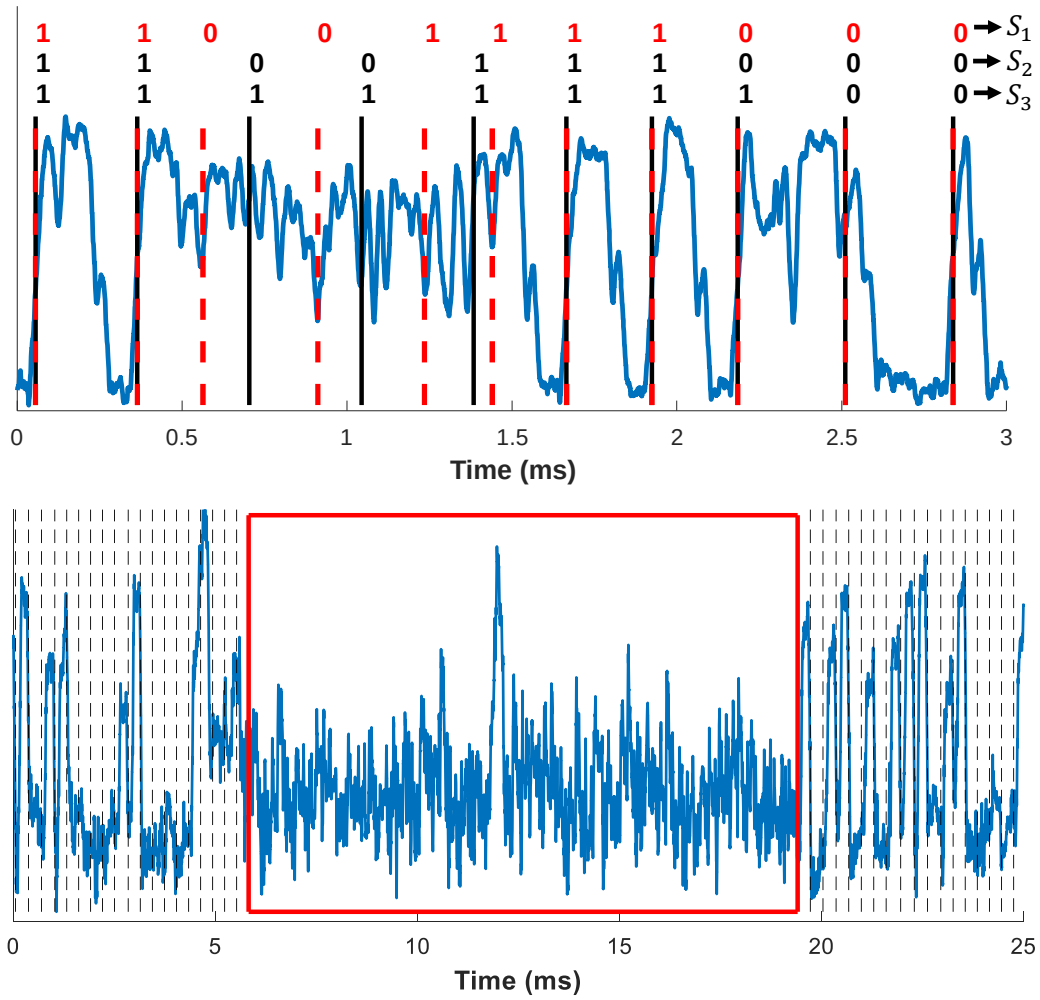


Figure 7.8: Bit deletion (top) and insertion (bottom) in the covert communication channel due to variations in signal timing.

4) Bit Deletion/Insertion

This covert channel presents many challenges not only because its signaling periods are not explicitly synchronized, but also because of occurrences of other system activity, such as interrupts and micro-architectural events (e.g., page faults, cache misses). These events can cause errors in the signaling periods they occur in, and also, in a sort of "domino effect", make errors much more likely for other signaling periods in the same batch.

An example of the bit deletion and insertion is shown in Figure 7.8. In the figure (top), the blue line represents the received signal, black lines correspond to the estimation of the

detection algorithm for the possible bit starting location. The dashed black lines (bottom) represent the starting location of a bit signal and the red box contains the region where the insertion occurs due to an interrupt. The actual bit starting points are given with red dotted lines (top). We observe that one of the bits is deleted. We also provide three sequences labeled with S_1, S_2 and S_3, which correspond to an actual transmitted bit sequence, actual transmitted bit sequence after deletion, and the estimated bit sequence, respectively. Here, the receiver's detection algorithm fails because the edges at the beginning of each transmitted bit completely disappear. The reason behind this disappearing is that other system activities get activated which suppresses the effect of the designed micro-benchmark. However, we observe that the deletion probability of the system is pretty low ($<0.2\%$). Therefore, this problem can be addressed by employing even relatively simple error correcting codes in the "transmitter" application. In our experiments, we use a very simple (parity) code, which keeps our "transmitter" application simple enough to manually implement on a target machine in a few minutes.

7.4.3 Experimental Evaluation

Measurement Setup

To show the feasibility of exploiting this covert channel, we present our experimental results in two practically relevant scenarios: when a compact and stealthy receiver apparatus is placed in close proximity to the target system, and when a larger (briefcase-sized) antenna is placed up to 2.5 meters away, and also when the antenna is in an adjacent room, 1.5 m away but with a structural 35 cm thick wall included in that distance.

We used a software-defined-radio (RTL-SDRv3 [165], commercially available for $25) for signal acquisition which is as large as a small flash drive. For proximity measurements, we used a coin-shaped handmade 33-turn coil magnetic field probe with a radius of 5 mm which costs <$5 (no amplifier is used). For the distance and non-line-of-sight (NLoS) measurements, we used the same SDR with a magnetic loop antenna (AOR-LA390 [166])

with a radius of 30 cm which costs $200, including a built-in 20dB amplifier.

For the *transmitter*, we used $usleep()$ for UNIX-based machines, and $Sleep()$ for Windows-based machines. The sampling-rate for the SDR was set to 2.4 million samples per second, which is the maximum this SDR is capable of. We used 1024 point FFT with maximum overlapping. The receiver's detection algorithm was implemented in MATLAB 2017-b. For synchronization between the transmitter and receiver at the start of the communication, the transmitter sends a pre-defined bit-stream of interleaved ones and zeros followed by a known short bit-stream of zeros only. The transmitter then sends a *preamble* to indicate the start of the transmission, and then sends the actual data. Depending on the requirement, the data can be sent in packets or continuously. Unless otherwise indicated, we used `SLEEP_PERIOD`$= 100\mu s$ (for UNIX-based machines) or $= 1ms$ (minimum possible for Windows-based machines) and set `LOOP_PERIOD` such that the active and idle periods have (almost) equal lengths.

We used 6 laptops from 5 different vendors (see Table 7.1), various processor architecture generations, and three popular OS families (Linux, MacOS, and Windows). To receive the EM signals, we placed the probe 10 cm away from the computer. To find the position where the signal power is the strongest, we manually localized the source of the signal. We found that the position which the signal is strongest is slightly different from one laptop to another but they are mostly concentrated in the middle and/or the bottom right quarter of the laptop (on top of the keyboard). Note that all these measurements were done without making any changes to the laptop's package. Also, all the measurements were done in the presence of other system's normal activities (i.e., handling interrupts, context-switch, etc.).

Near-Field Measurements

To measure the BER and bit-rate, we created a randomly-generated sequence of bits. Also, to decrease the BER, the *transmitter* application inserts parity bits such that the minimum Hamming distance between different codewords was at least three (to correct one error).

Table 7.1: List of laptops and their OS and (Intel) processor architecture used in our experiments.

Model	OS	Architecture
Dell Precision 7290	Windows 10	Kaby Lake
MacBookPro-2015	macOS (Mojave)	Broadwell
Dell Inspiron 15-3537	Linux (Debian)	Haswell
MacBookPro-2018	macOS (Mojave)	Coffee Lake
Lenovo Thinkpad	Linux (Ubuntu)	SkyLake
Sony Ultrabook	Windows 8	Ivy Bridge

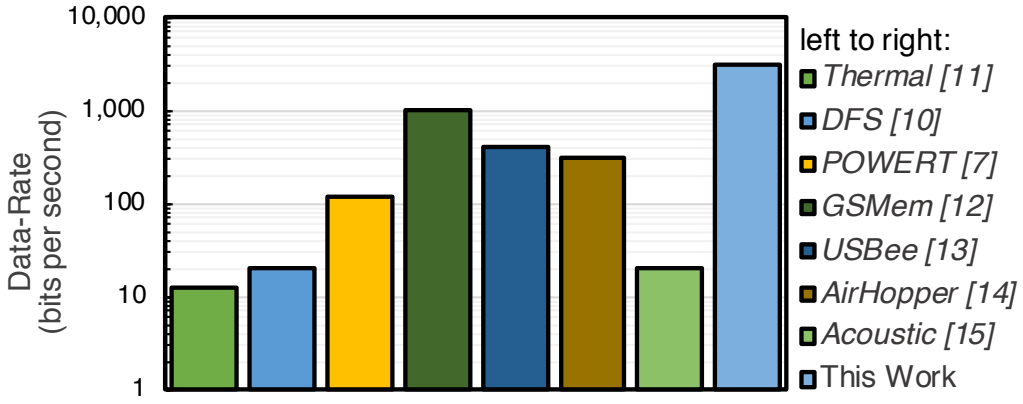


Figure 7.9: Transmission-rate comparison (higher is faster) between the proposed covert channel and the state-of-the-art (shown in log-scale). Each bar represents an existing attack. The proposed work achieve more than 3x higher TR compared to the fastest attack.

In Table 7.2, we provide the experimental results for six laptops. The columns of the table correspond to the average number (for 5 runs) of BER, transmission rate (TR), insertion probability (IP), and deletion probability (DP). To calculate the IP and DP, we compared the actual transmitted sequence with the received sequence based on the algorithm described in Section 7.4.

As shown in the table, the proposed covert channel can achieve up to 3.7 kbps (kilo-bits per second) while having less than 0.1% BER with a low-cost and compact setup, and that the main determinant of the bit-rate is the operating system, i.e., the precision with which the *transmitter* application can control idleness time - the `usleep` in Linux and MacOS is significantly more precise than `sleep` used in Windows, so the TR in Linux and MacOS

Table 7.2: Experimental results for close proximity. The results include the bit-error-rate (BER), transmission-rate (TR), insertion probability (IP), and deletion probability (DP) for the proposed covert channel on different laptops.

	BER	**TR (bps)**	**IP**	**DP**
DELL Precision	2×10^{-3}	982	0	0
MacBookPro (2015)	3×10^{-2}	3700	0	3×10^{-3}
DELL Inspiron	8×10^{-3}	3162	4.5×10^{-3}	6.3×10^{-3}
MacBookPro (2018)	2.8×10^{-2}	3640	0	2.9×10^{-3}
Lenovo Thinkpad	5×10^{-3}	3020	0	1×10^{-3}
Sony Ultrabook	4×10^{-3}	974	0	5×10^{-3}

systems is 3-4 kbps while the TR for Windows systems is slightly below 1 kbps.

Figure 7.9 compares the maximum TR of the proposed covert channel to the state-of-the-art. Specifically we compared our method to 7 different attacks that leveraged *physical* side-channels to establish a covert channel. As can be seen from the figure, the proposed covert channel can achieve more than 3x faster TR compared to the fastest existing covert channel attack, GSMem [150]. Note that, to provide a *fair comparison*, for each work we only report the TR with a similar measurement setup (if available), i.e., similar distance and/or measurement equipment.

To study the effect of **background activity** on the TR, we repeated our measurements, this time with adding a resource-intensive background activity (in addition to normal OS background activities which were present in all the results showed in Table 7.2 and Fig-

Table 7.3: Experimental results with distance. The results include the bit-error-rate (BER), transmission-rate (TR), insertion probability (IP), and deletion probability (DP).

Distance	BER	TR (bps)	IP	DP
1 m	9×10^{-3}	1872	5×10^{-3}	0
	9×10^{-4}	1645	1×10^{-3}	0
1.5 m	5×10^{-3}	1454	0	0
2.5 m	8×10^{-3}	1110	0	0

ure 7.9). We observed that to handle background activities, the OS tends to produce short bursts of activity which do not affect our covert-channel detection much since they are smaller than one sleep/active period. Longer bursts of activity do create some errors in our detection, but these are fixed/corrected using parity-bits (c.f. §7.4.2). However, if there are longer periods of activity, e.g., intense activities by other processes, the covert-channel transmission can either lower the transmission-rate or even pause temporarily. Using this new measurement, we found that to achieve similar BER to that of Table 7.2 in the presence of resource-intensive background activity, the TR has to be decreased (only for UNIX and macOS laptops), on average, by 15% (with worst-case of 21%).

Distance Measurements

LoS Measurements. To study the effect of distance on TR, we computed the BER and TR while the loop antenna was put 1, 1.5, and then 2.5 meters away from the target laptop. We manually set the antenna's orientation to maximize the signal SNR. As mentioned in §7.4.3, the loop antenna has about a 30 cm diameter and it can be easily hidden in a briefcase.

Table 7.3 shows the results for these 3 distances for the DELL Inspiron laptop. As can be seen from this table, the TR can be approximately 2 kbps when the distance for the communication link is around 1m. Additionally, we decrease TR so that BER of the system at different distances is almost the same for a fair evaluation of the performance of

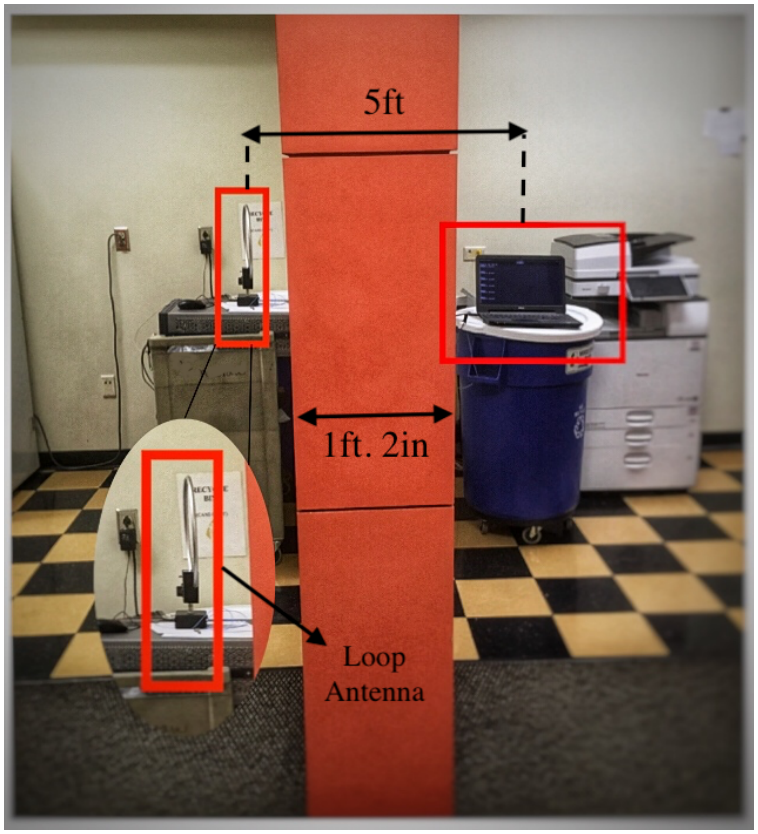


Figure 7.10: Experimental setup when an attacker and a victim are separated by a wall.

the communication link.

It can be observed that having smaller TR for larger distances makes the communication more reliable. Although the rate decreases as the distance increases, we can receive 1110 bps when the distance is about 2.5m.

NLoS Measurements (through the wall). The setup for NLoS measurements is shown in Figure 7.10 where the transmitter and receiver are separated by an office wall which has about 35 cm thickness. Moreover, as can be seen, there are other electronic devices such as a printer in the transmitter's room and a refrigerator in the receiver's room which also generates unintentional EM emanations which can interfere with the laptop's emanations and hence makes the signal noisier. Note that we intentionally chose this setup to show that the proposed covert communication can work reliably even in the presence of other sources

of EM emanations.

To maintain a low BER compared to the near-field measurements, the bit-rate had to be decreased to 821 bps while having 6×10^{-3} BER. However, as the signaling period is now significantly longer than the typical duration of an interrupt, the detection algorithm is better able to tolerate these system events, and so deletion and/or insertion of bits occurs less often or not at all. Overall, the NLoS measurements show that the covert communications are still possible (although at the lower rate) even when the transmitter and receiver are in two separate rooms which makes the attack more stealthy.

7.5 Keylogging

7.5.1 Overview

The goal in keystroke logging, or *keylogging*, is to find when and which key has been pressed on a computer keyboard. This, in turn, can lead to stealing sensitive information, passwords, etc. In general, every keystroke can be shown as a 3-tuple [167], (t_p, t_r, k), where t_p is when the key is pressed, t_r is when the key is released, and k is the physical key identifier. Using this definition, keylogging becomes a two-phase problem: *keystroke detection* (i.e., correctly finding t_p and t_r) and *key identification* (i.e., finding k). It is important to mention that while, ideally, the goal for keylogging is to find the exact character for each keystroke, realistic attacks [167] typically provide *a significantly reduced possible states* for a keystroke/word. Using this reduced state space, an attacker can then leverage a *brute-force* attack to eventually find the actual characters. In this section, we show how EM emanations from PMU can be leveraged to provide highly accurate keystroke detection.

Compared to the existing works that leveraged digital (e.g., cache usage) [168, 169, 170, 171] and/or physical [51, 59, 172, 173, 174, 175, 176] for keylogging, the main advantage of using the proposed side-channel for keylogging is that it enables the attackers to perform an attack from a distance behind a wall even on an isolated air-gapped computer.

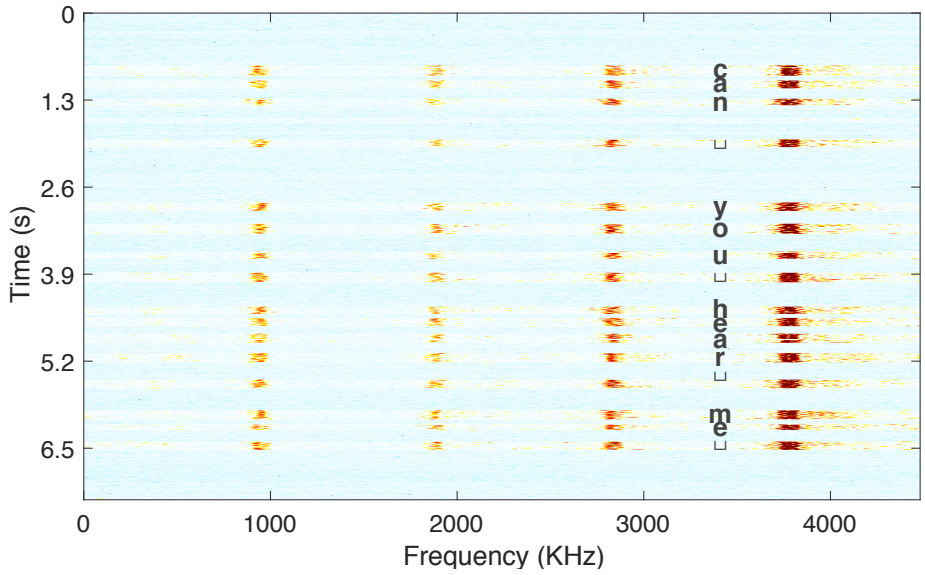


Figure 7.11: PMU's EM emanations over time when the user is typing: "`can you hear me `".

7.5.2 Keylogging Side-Channel Attack

The main idea behind this attack is that pressing a key creates a *burst* of activity on the processor which, in turn, causes the (otherwise idle) processor to briefly switch to an *active* state, hence creates (stronger) EM emanations from the PMU.

Figure 7.11 shows the spectrogram for the EM emanations received from the PMU (using the setup used in §7.4.3) when the user is typing a sentence/password: `can you hear me`. As can be seen in the figure, each character (including the white-spaces shown as '␣') has a distinguishable pattern which means the total number of characters can be found with high accuracy using a simple-yet-effective detection algorithm. Moreover, as can be observed in Figure 7.11, the number of words and their length can also be inferred by grouping relatively close spikes (lines in the spectrogram) together.

Apart from the number of characters, words, and the length of each word, existing work [177, 178] has shown that the duration of each keystroke and the time difference between two consecutive keys can also be leveraged to further reduce the search space for key identification. Salthouse [177] has empirically shown that *(i)* keys that are far apart are

pressed in quicker succession than keys that are close together (e.g., see `you` vs. `can` in Figure 7.11). *(ii)* letter pairs that occur frequently in language are typed in quicker succession than infrequent letter pairs (e.g., see `hear` in Figure 7.11). *(iii)* practicing a specific keystroke sequence can significantly reduce inter-key timings (e.g., compare the timing for the white-space character in the first word, `can`, vs. the rest of the sentence). Using these findings, after properly detecting each keystroke's timing, supervised or unsupervised classifiers can be used to identify the keystrokes/words and/or to reduce the search space.

7.5.3 Experimental Evaluation

Setup. We used the same setup discussed in subsection 7.4.3 for near-field and distance measurements and performed our measurements at three distances: 10cm, 2m, and 1.5m through the wall while the DELL Precision laptop was used. For each distance, a randomly-generated text with 1000 words[4] is typed (by the same person) in the Chrome browser while the signal was recorded. It is important to mention that all the measurements were done in the presence of other system's normal activities, including activities related to the browser.

Keystroke/Character Detection. To find the total number of characters (i.e., total number of keystrokes), we first normalized and then transformed the signal into a sequence of (non-overlapping) spectral samples (SS) by using short-time Fourier transform (STFT), which divides the signal into equal-sized segments (*windows*) each 5ms long. STFT then applied the Fast Fourier Transform (FFT) to each window to obtain its spectrum. We then chose a frequency band which contains the spikes related to PMU (the band is typically known for each device, but can also be easily found using standard peak detection techniques). We then used a simple thresholding technique (cf. §7.4.2) to decide whether the particular window contains a keystroke or not. To reduce false-positives, our detection algorithm then filtered out keystrokes that are shorter than a threshold (i.e., 30 ms in our experiments)

[4]obtained from `https://www.livechatinc.com/typing-speed-test/#/`

Table 7.4: Experimental results for keylogging in different distances. Results show the accuracy of correctly detecting characters and word lengths.

Distance	**Char. Acc.**		**Word Acc.**	
	TPR	FPR	Precision	Recall
10 cm	100%	3%	71%	100%
2m	99%	1.8%	70%	100%
1.5m (with wall)	97%	0.7%	70%	98%

knowing that a valid keystroke should take longer than this threshold.

Table 7.4 shows the accuracy (in terms of true-positive and false-positive rates, denoted as TPR and FPR) of detecting characters for the three distances. False positives are mainly caused by other system activity, such as handling of the browser requests, which also appeared as a (typically much shorter) bursts of activity. As the distance increased, such activity became less prominent (i.e., lower signal amplitude) which, in turn, caused a reduction in FPR. However, this reduction in the amplitude also affected the TPR as the emanations caused by keystrokes also became weaker.

Word Detection. Once characters are correctly detected, the next step is to group the characters into the words. There are numerous techniques to reconstruct words based on individual characters. Particularly, we used the method proposed by Berger *et al.* [174]. Table 7.4 shows the accuracy. Since word-length is a multi-class classification problem, instead of TPR and FPR, we report *precision* percentage as the fraction of correctly labeled words (i.e., words with correctly predicted length) among the retrieved words, and also report *recall* as the fraction of words that have been retrieved over the total amount of existing words in the text. As the table shows, we were able to detect almost all the words while keeping the precision accuracy above 70%. The distance did not have a significant impact on either of these two metrics.

7.6 Prior Work and Countermeasures

Electromagnetic Side-Channels. Prior work on EM side-channel mainly focused on the extraction of small amounts of highly sensitive data (such as cryptographic keys) from the system [35, 42, 179, 180, 181, 42]. Beyond extracting sensitive data values, EM emanations have also been used to learn more about program behavior, e.g., for identifying web pages during browsing [182], program execution profiling [127, 183], finding anomalies in software activity [184, 105, 185, 129], establishing trust [186], etc.

While existing works have shown the existence of side-channel EM emanations from different electronic components within a computer, to the best of our knowledge, this is the first work that identifies the EM-based vulnerability created by the power management unit and exploits this vulnerability to establish a covert channel and/or a keylogging framework.

Also related to this work, are the methods based on the variations on the voltage and their corresponding EM emanations [37, 45, 57]. The main difference between the proposed side-channel and these works is that voltage-variation is *instruction-dependent* while the VRM side-channel is a *micro-architecture vulnerability*. Note that while the EM emanations for the voltage-variation are the strongest close to the VRM (because VRM spends most of the power) its existence is unrelated to how the VRM operates (i.e., the created side-channel is application-dependent, not VRM-dependent). Due to this difference, the proposed side-channel creates much stronger signals which can be received several meters away. Moreover, unlike other methods, it is resistant against techniques like randomization/blinding.

Physical Side-Channels. Exploiting other physical side-channels such as acoustic, temperature, etc. [43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60] to create a covert channel/keylogging, is another groups of related work. Overall, depending on the setup, different physical side-channels can be used. The main differences between them

are *a)* measurement requirements (i.e., cost and size of the receiver, signal-to-noise ratio and maximum achievable distance, etc.), and *b)* the maximum achievable bit-rate which indicates how severe and stealthy the attack could be. As shown in §7.4.3, compared to the state-of-the-art, the proposed side-channel can achieve more than 3x faster bit-rate mainly because it relies on the fast switches between *idle* and *active* states on PMU.

Yet another related work to this paper are the methods to exploit the power management unit by leveraging *digital* side-channels [148, 149, 29, 187]. Most recently, Khatamifard *et al.* [149] proposed POWERT, a new method to leverage the power budget for creating a covert channel. To transmit data, POWERT either creates power-intensive activities or remains idle. To infer the transmitted data, the Sink process (also located on the same physical device), measures its own performance by running a known application. The key idea is that due to the limitation in the power budget, the Sink's performance is directly impacted (modulated) by the source activity. In this paper, we showed that compared to POWERT [149], our proposed covert channel can achieve significantly higher data-rate (>20x) since it does not rely on the indirect measurement of the performance while it is less susceptible to the noises generated by the system. Moreover, as mentioned in section 7.2, the threat model for these attacks are different from the one used in this paper since they are leveraging digital side-channels.

Circuit-Level Methods. Another related body of work to this Chapter are circuit-level techniques to improve the EM/Power side-channel resistance, especially for integrated cryptographic accelerators on the chip such as AES engines [188, 189, 190, 191]. These techniques are mainly focused on using on-chip integrated voltage regulators [192] (hence reducing the amount of pin accessibility to the attacker) and adding randomness to the voltage regulator which, in turn, creates random patterns in the side-channel signals [193].

Countermeasures. There are multiple possible countermeasures to mitigate the discovered

vulnerability. At the system-level, the system software and/or user can disable P/C-states to perform highly-sensitive computation (briefly, because the energy overheads are significant and/or because of the temperature-control requirements). Another solution in this vein would be to use methods like architecture-blinking [194] (processor is briefly disconnected from the PMU during sensitive computation to avoid PMU-related leakage of information). A more fundamental solution, however, would involve adding pre-determinism, randomness, and/or noise to the operation of the PMU which, mainly, should be performed at the circuit-level. Finally, traditional EMI-shielding methods can be used to reduce the SNR and increase the BER (with their own limitations/overheads).

7.7 Conclusions

In this Chapter, we presented a new, previously unexplored, physical side-channel vulnerability caused by the design, implementation, and typical use of the power management unit (and its main component, the voltage regulator module) in modern computers. Specifically, we showed that different power states on the system (which are primarily utilized to optimize the energy efficiency) and the way existing PMUs create these states can lead to a side-channel which can leak sensitive information about the current state of the system.

To exploit this side-channel, we demonstrated two real exploits: *a)* a covert channel which leveraged an inexpensive and compact setup to establish the covert communication and *b)* a keylogging framework which can accurately detect the keystrokes even when the receiver was behind a wall on a separate room.

In our experimental evaluation, we used a number of laptop systems, from various vendors, as the target system. We also performed our measurements under realistic conditions that include environmental EM noise and having other applications run on the target system concurrently with our data-exfiltration applications.

Our results showed that this newly-discovered side-channel exists on all systems we evaluated, regardless of hardware platform, operating system, and vendor. Further, we

showed that, compared to the state-of-the-art, exploiting this side-channel can lead to a much faster covert channel (when a physical side-channel is used).

We believe that our discovery and detailed analysis of this new side-channel vulnerability will help raise professional awareness and academic awareness about security implications of power management, and help existing and future generations of computers should address this vulnerability carefully, especially when the system is used in a critical environment.

CHAPTER 8

EMSIM - A MICROARCHITECTURE-LEVEL SIMULATION TOOL FOR ACCURATELY MODELING ELECTROMAGNETIC SIDE-CHANNEL SIGNALS

8.1 Abstract

Physical side-channel attacks have become a serious security concern for computing systems, especially for embedded and Internet-of-Things devices, where the device is often located in, or in close proximity to, a public place, and yet the system contains sensitive information. To design hardware and/or software that is highly resilient to such physical side-channel attacks, there is a need for accurate and efficient design-stage quantitative analysis of side-channel leakage. For many system properties (e.g., performance, power, etc.), cycle-accurate simulation can provide such an an efficient-yet- accurate design-stage estimate. Unfortunately, for an important class of physical side-channels, electromagnetic (EM) emanations, an accurate and efficient simulation model does not exist, and there has not even been much quantitative evidence about what level of modeling detail would be needed for high accuracy, and/or how much accuracy is sacrificed when not modeling certain aspects of the hardware, underlying physics, and/or microarchitecture.

While there are tools to identify leaks and metrics to calculate EM side-channel leakages, they are limited due to three main reasons: First, they are mainly focused on developing metrics to estimate the leakage rather than simulating the original analog signal. Unfortunately, relying only on these metrics rather than analyzing the actual physical signal may not be sufficient since they may not reveal the entire true mutual information. Secondly, existing methods only model the system at architecture-level, i.e., associating a (leakage) value to individual instructions based on the ISA (e.g., depending on the previous state of the system, values, etc.), and ignore the micro-architecture events and activities of

the system such as pipeline stages, stall cycles, cache miss, branch mis-prediction, etc. on the signal. Unfortunately, these approximations can lead to a significant inaccuracy since these events may have noticeable effect on the EM analog signal thus can leak extra information about the application. In other words, by staying at the ISA-level, the model neglects to account for how that instruction interacts with other instructions and the underlying hardware. Third, due to neglecting the impact of micro-architecture, these methods assume the entire hardware as a single source, and then only model the EM emanations based on this single source. Unfortunately, such an assumption can lead to large inaccuracies since different micro-architecture components (e.g., cache, register-file, etc.) generate different electromagnetic waves with different polarization and/or phase, and hence, they may have constructive or destructive impacts on each other and the overall received signal.

To address these challenges, we will develop a new method that enables simulation of the EM side-channels cycle-by-cycle using the detailed micro-architectural model of the device. We then will quantitatively assess the accuracy of simulator-generated signals by comparing them to actual EM signals collected while an actual processor executes the same program code, and we will further identify how much accuracy suffers when key micro-architectural features and hardware behaviors are omitted from the simulation model.

The outcome of our work will be an open-source side-channel simulator tool and can be used for leakage estimation. To the best of our knowledge, this is the first micro-architecture-level EM side-channel model, which allows it to provide much more precise estimates of leakage not only for individual instructions, but also for sequences of instructions (when the goal is to assess and improve leakage from a particular piece of code on a set of hardware platforms), and also the first model that can asses leakage from a particular part of the processor and/or the memory system (when the goal is to make the hardware design less “leaky”) while maintaining the performance advantages of a cycle-accurate simulation relative to a physics-based model.

8.2 Background

Information leakage through side-channels has become a serious concern in securing computing systems that are located where potential attackers may gain enough physical access to carry out the attack [42, 49, 195, 196, 197, 198]. This is especially a problem for embedded and Internet-of-Things (IoT) systems which often contain sensitive data, such as sensor data, login information for over-the-network management of the system and/or accessing back-end cloud infrastructure, and are often placed in publicly accessible locations. For some side-channels, such as electromagnetic (EM) emanations, physical proximity can be leveraged to attack systems that are considered to be physically secure but are located near publicly accessible locations, e.g., in-wall "smart building" sensors, security cameras, etc. [37, 38, 180, 131, 199].

To design these systems to be highly resilient to side-channel attacks, side-channel leakage needs to be quantitatively assessed and attributed to specific hardware and software components, relatively early in the design of the system. For other system properties such as performance, power consumption, reliability, etc., such early quantitative assessment and attribution typically relies on cycle-accurate simulation [92, 147, 200, 201, 202, 203, 204, 205] that models the relevant activities in the system with enough detail to achieve good accuracy, but without modeling most of the low-level activities that would render the simulation too slow to be useful for evaluating relevant scenarios. If such efficient-yet-highly-accurate simulation would exist for EM emanations, hardware designers and architects could include EM side-channel leakage among their design considerations [194, 206, 207, 208, 209, 210], compilers could use simulation models to optimize for reduced leakage [211, 212, 213], software designers could detect and mitigate information leakage problems for security-sensitive applications [214, 215, 216], etc.

Unfortunately, for assessment and attribution of EM side-channel emanations, no such efficient-yet-highly-accurate simulation exists, and there has *not* even been much quanti-

tative evidence about what level of modeling detail would be needed for high accuracy; how much accuracy is sacrificed when not modeling certain aspects of the hardware and hardware-software interaction; whether high accuracy for estimating EM emanations can be achieved without modeling the underlying physics - how magnetic and electric fields change in response to current flows through semiconductor and metal structures that form the circuitry of a system; whether it is sufficient to model ISA-level properties without modeling the microarchitecture; etc.

While there are some tools and metrics to quantify EM side-channel leakages [217, 36, 218, 219, 220, 221, 222, 223], ***they are limited due to three main reasons***: *First*, they are mainly focused on developing metrics to estimate the information leakage itself, i.e., mutual information between the signal and program secrets, rather than modeling the actual analog signal. Relying only on these metrics rather than analyzing the actual signal, may not be sufficient since these metrics inherently make assumptions about the aspects of the signal that attacker may exploit, i.e., they may not reveal *all* of the information the signal may contain. *Secondly*, existing methods only model the system at *architecture-level*, i.e., associating a (leakage) value to individual instructions based on the ISA, and ignore the micro-architecture activities such as pipeline stages, stall cycles, etc. on the signal. As we will show in this paper, this can lead to significant inaccuracy, mainly because the model, by staying at the ISA-level, neglects to account for how that instruction interacts with other instructions and the underlying hardware. *Thirdly*, by neglecting the impact of micro-architecture, these methods implicitly assume that the entire hardware design is a *single source* of information, and then only model the EM emanations based on this single source. Such an assumption can lead to large inaccuracies since different micro-architecture components (e.g., cache, register-file, etc.) generate different electromagnetic waves with different polarization and/or phase, and hence, they may have *constructive or destructive* impacts on each other and the overall received signal.

To address these challenges, in this work we take a different approach. We develop a

simulator, *EMSim*, that is able to simulate the EM side-channel signals cycle-by-cycle using the detailed micro-architectural model of the device. We then quantitatively assess the accuracy of simulator-generated signals by comparing them to the EM signals collected while an actual processor executes the same program code, and we further identify how much accuracy suffers when key micro-architectural features and hardware behaviors are omitted from the simulation model. By using a systematic set of measurements, along with *regression-based parameter estimation and multiple-input-single-output (MISO) communication system modeling*, we show that *EMSim* is able to produce an EM signal that closely matches the actual signal, and it is robust against different sources of variability.

8.3 Experimental Methodology

Before describing how *EMSim* simulates side-channel signals, this section describes our experimental methodology for obtaining EM signals generated by actual hardware, as this information may help the reader better understand the discussion of *EMSim*.

8.3.1 Hardware and Measurement Setup

While signals can be collected from an actual off-the-shelf processor, such a processor would be difficult to model in detail because many of its microarchtiectural details (of even entire microarchitectrual blocks) are not publicly available, and unceratinty about how well the model used in *EMSim* matches the actual miscroarchitecture would make direct comparison of real and simulated signals difficult. Therefore, we implement a RISC-V [224] processor on an FPGA (using Verilog), giving us full knowledge of the actual microarchitecture of the processor.

We implement a 32-bit *base* RISC-V ISA [225] which provides a convenient ISA and software toolchain "skeleton" suitable for resource-constrained design scenarios such as embedded systems [224]. We also implement the multiplication/division extension ("M") to support these activities. Prior to using the processor for our measurements, we ex-

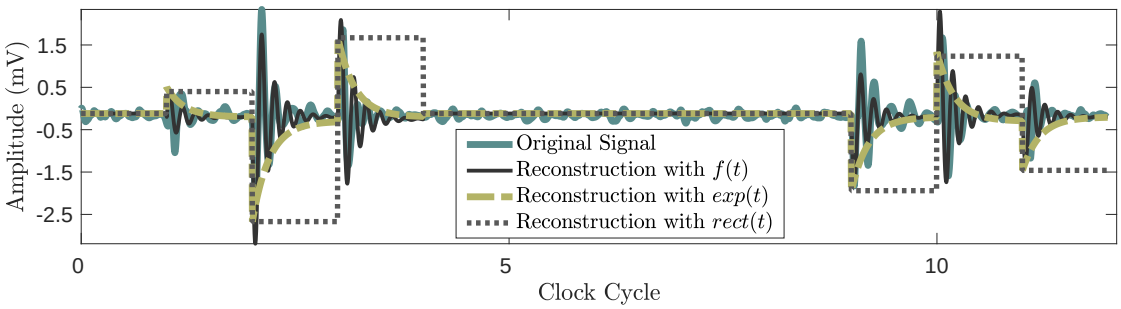


Figure 8.1: Reconstructing the original signal using three different approaches. Using a combination of a sinusoidal and an exponential function ($f(t)$ in Equ. 8.5) can achieve the best accuracy.

tensively tested the correctness of the processor's implementation. The designed processor has five pipeline stages namely: `Fetch`, `Decode`, `Execute`, `Memory`, and `Writeback`. The processor also has **(i)** a branch prediction unit with a *2-level predictor* [226] and a *branch-target-buffer* (BTB), **(ii)** a 32-entry 32-bit *register-file*, and **(iii)** a 32KB cache. *Cache-hit* takes one extra cycle and reading from memory takes extra 2 cycles (in addition to the cache latency). However, these delays can be changed, e.g. to study their effect on the side-channel signal.

Using the implemented processor, we then use a magnetic probe and a signal acquisition device (an oscilloscope) to receive and record the EM signals (more details on §5).

8.3.2 Signal Acquisition

Capturing the emanated signal is the first step to model the signal. Unfortunately, measuring the ideal emanated signal (i.e., not corrupted by additive channel white noise) is not possible if only one-time-run of any instruction sequence is considered. One option is to collect many one-time-run signals and take the average. The problem with this approach is capturing synchronized signals, i.e., the starting points of captured signals may not correspond to the same processing-time of the given instruction sequence. To address this problem, we use a novel signal-processing method called "*modulo operation*" [227] to cre-

ate a highly accurate estimate of the ideal emanated signal, i.e., to remove most of the noise and distortion that was present in the actual signal due to under-sampling, synchronization, noise, and other imperfections that are unavoidable during practical collection of signals.

The main parameters for the *modulo operation* are the number of clock cycles to execute a given sequence,`noc`, the sampling-rate of the instrument, and the clock frequency of the device. After having these parameters, the next step is to collect the emanated signal. For that, a given sequence is executed several times (1000 times in our measurements). Each set of measurements consists of a sequence of instructions (called `sequence`). The goal is to retrieve the emanated signal for the `sequence`.

The next step is to utilize the *modulo operation* to map each received samples to average these many measurements. Assuming T_s is the total time to execute the sequence once (i.e., $T_s = \texttt{noc} \times T_{clk}$), it applies the following operation to the sampling time of each sample to map each sample to its fundamental period:

$$\Delta_m = mod(T_m, T_s), \tag{8.1}$$

where T_{clk} is the clock time, Δ_m is called the *modular offset*, and T_m is the sampling time of m^{th} sample. After obtaining the modular offsets for each sample, the *modulo operation* takes the mean of the samples that have same modular offset, to produce the desired signal. Further signal-processing techniques such as moving average, Gaussian filtering, etc., can be applied to this generated signal to obtain smoother reference signals.

8.3.3 Signal Reconstruction

Simulating an analog signal can be considered a signal-processing reconstruction problem where a continuous signal (i.e., EM in this case) needs to be determined from a sequence of equally-spaced samples with sampling-rate T, where T is preferably much smaller than T_{clk}. Such a signal can be ideally reconstructed using Whittaker-Shannon interpolation

formula [228], however, it is well-known that such a method is not feasible in practice. Instead, a popular method for signal reconstruction is zero-order hold (ZOH) technique where a continuous signal, y, can be reconstructed from a sample sequence $x[n]$, assuming one sample per time interval is T:

$$y(t) = \sum_{n=-\infty}^{+\infty} x[n] \times rect(\frac{t - T/2 - nT}{T}). \tag{8.2}$$

To improve the ZOH accuracy, in this paper we make an observation that *switching activity in a processor is synchronized to its clock* and most of the switching happens right after the positive/negative edge of the clock. Thus, instead of using a rectangular function (which implies that activity is evenly spread over a cycle), a decaying function can be used:

$$f_1(t) = e^{-\theta t}\mathtt{u}(t), \tag{8.3}$$

where θ is a positive normalization factor that changes the width of the signal, and $\mathtt{u}(t)$ is the unit step function. Substituting $rect()$ in Equ. 8.2 by the exponential we get:

$$y(t) = \sum_{n=0}^{+\infty} x[n] \times e^{-\theta(t-nT)} \times \mathtt{u}(t - nT). \tag{8.4}$$

However, we observed that the received signal is also exposed to oscillations with decreasing magnitude. To meet the requirements for both oscillations and decreasing amplitude, combining sinusoidal with exponential can increase the accuracy further:

$$f(t) = \sin(2\pi t/T_0) \times e^{-\theta t} \times \mathtt{u}(t), \tag{8.5}$$

where T_0 is the periodicity of the sinus function. Again, substituting $rect()$ in Equ. 8.2 by

$f(t)$, we will have:

$$y(t) = \sum_{n=0}^{+\infty} x[n] \sin(\frac{2\pi(t-nT)}{T_0}) e^{-\theta(t-nT)} \mathtt{u}(t-nT). \tag{8.6}$$

In Figure 8.1, we plot a measured signal and its reconstructions with these options. We observe that $f(t)$ explains the behavior of the received signal much better. Thus, by finding $x[n]$ for each cycle and using Equ. 8.6, the analog side-channel signal can be modeled.

8.4 EMSim Modeling

8.4.1 Overview

Fundamentally, EM side-channel signals are created due to *bit-flips* at the transistor-level [229, 38]. In principle, all transistors and metal-layer interconnect components contribute to the signal, thus the signal could be modeled using all transistors and on-chip wires as predictor variables, which *should* be highly accurate but is practically infeasible. As a result, the main challenge in model-building is to select (or discard) potential predictors in a systematic manner, to achieve a trade-off where feasibility (or even efficiency) is achieved without a major sacrifice in accuracy.

To achieve a simple yet accurate model, existing methods are mainly focused on individual instructions and their operands, and they attribute an "average" behavior to each of the instructions, rather than model its cycle-by-cycle effect on the processor's hardware. In effect, these methods model a simplified *one-instruction-at-a-time* implementation, essentially ignoring pipeline effects and other important aspects of the micro-architecture.

To more accurately simulate EM signals, we model micro-architectural components as ***independent*** sources of EM emanations and then further group these units in each pipeline stage as an individual source. We used pipeline stages as the sources mainly because we observed that each instruction has different footprint in each cycle, and the side-channel generated at each cycle is a combination of these activities in *ALL* stages. Using this

methodology, we model a multi-input (pipeline stages), single-output (EM signal) system (MISO).

Leveraging this approach, the **challenges** are ***a)*** *how to model the signal for individual sources*, and ***b)****how to properly combine the signals* generated by each source to accurately form the side-channel signal.

8.4.2 Signal Amplitude for Individual Sources

In practice, there are two contributors in creating EM side-channel signals for each pipeline stage. The first group of contributors, which we call *instruction-dependent* activities, are caused by the switching activities of micro-architectural units (e.g., register-file, ALU, etc.) that are utilized in that stage (e.g., whether the register-file is being written or not).

The second group, *data-dependent* activities, are created due to bit-flips on the data-bus, address-bus, and any other registers that hold operand's values. These bit-flips are independent from the instruction-type but are dependent to the previous state of the bus. In the following, we will describe how we *independently* measure each of these two groups.

Instruction-Dependent Activities. To independently measure these groups, we first minimize the effect of the *data-dependent* activities by setting all the operands, addresses, and immediate values to zero. This approach enables us to measure the *baseline* signal for each stage which is *only* created by the switching activities of the micro-architectural units used in that stage.

After decoupling the data-related activities from the signal, the second challenge is to minimize the effects of other stages on the generated EM signal. Recall that we mentioned *ALL* pipeline stages contribute to the overall signal, however, ideally we want to be able to measure the effect of each stage *separately* so that we can use them as the basic-blocks to reconstruct the overall signal. To achieve that, we use `NOP` [1] instruction as the *baseline*

[1] In our core, NOP is implemented as "ADD r0, r0, r0". Given that r0 is hardwired to 0, this instruction generates no register-file and/or ALU activity.

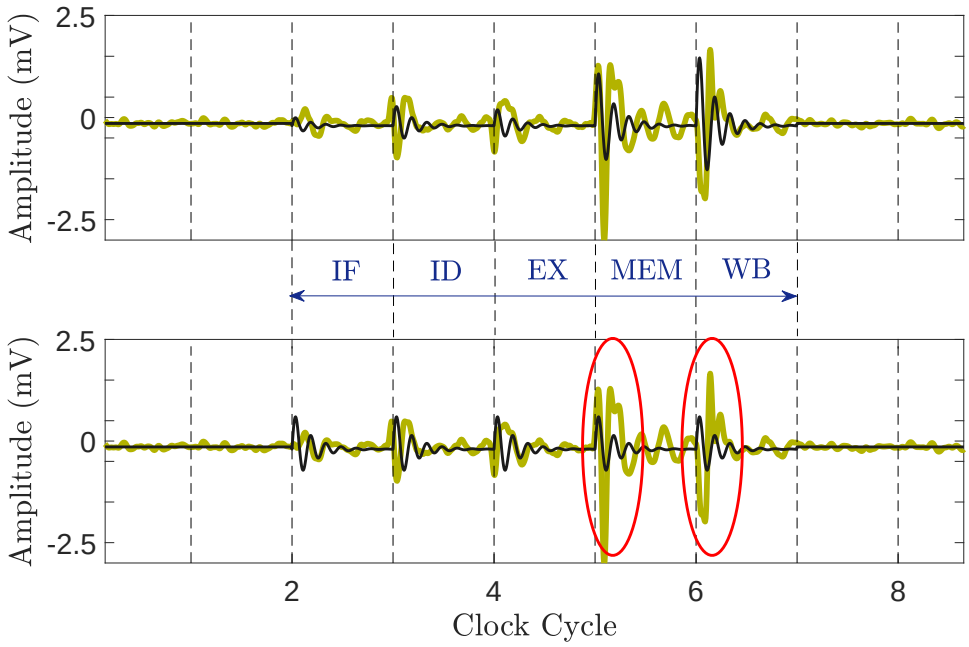


Figure 8.2: The signal amplitude for an `ADD` as it progress in the pipeline (while all other instructions are `NOP`). The actual signal is shown in light color (green). Darker color (black) shows the simulated signal when considering each pipeline stage as a separate source (top), and when considering the entire processor as a single source (bottom), and the largest differences between the two are pointed out using red ellipses.

since it has the minimum possible switching activity, and then create `NOP` $\rightarrow$ `inst` $\rightarrow$ `NOP` instruction sequence (for all instructions), while operands for `inst` are all set to $r1$ (and $r1 = 0$). Using this method, no data/operand-dependent bit-flips are created, but register-file, ALU, etc. may be used (depending on the instruction type). We then measure the signal amplitude for all instructions and every pipeline stages. We call this ***baseline hardware amplitude*** or A.

Figure 8.2 shows how the (actual) EM signal (shown in green/light color) changes as an `ADD` instruction progresses through the pipeline while all the other instructions are `NOP`. Using Equ. 8.6 and `NOP`$\rightarrow$ `inst`$\rightarrow$ `NOP` instruction sequence, we used our simulator to generate the signal. Further, to show why individual stages should be modeled separately, Figure 8.2 (bottom) shows the simulated signal when the "average" amplitude is used for all stages. As can be seen, failing to model each stage individually (as used in previous

work [221]) can lead to significant inaccuracies in some stages (note that using *max* instead of *average* also leads to similar inaccuracies).

Data-Dependent Activities. Once the baseline amplitude is measured, the next step is to find how this amplitude changes as the number of bit-flips changes due to value/operand used in the instruction and the previous state of the bus/register. Intuitively, the more bit-flips, the higher the amplitude should be thus we define *activity-factor*, α, as a ***scaling factor*** to the baseline activity, A. To find α, we first treat each bit-flip *equally*, and assume that each bit-flip has similar effect on the signal amplitude. We then calculate α as:

$$\alpha = 1 + \frac{(\mathit{flips}_{new} - \mathit{flips}_{base})}{\mathit{flips}_{total}}, \tag{8.7}$$

where flips_{new} is the total number of flips for the current instruction, flips_{base} is the total number of flips when previous instruction is `NOP`, and flips_{total} is the maximum possible number of flips for the current instruction. Using this equation, we then define $A' = \alpha \times A$, and use it to simulate the signal. Figure 8.3 (bottom) shows the original signal (shown in light green), and the simulated signal using this approach (shown in black) for the similar `NOP`$\rightarrow$ `inst`$\rightarrow$ `NOP` instruction sequence discussed in the previous section. As can be seen in the figure (bottom), this *"averaging"* modeling can not accurately predict the amplitude of the signal which indicates that *not all the bit-flips have the similar impact on the amplitude*. Our further investigation confirmed this theory. Particularly, we found that flips in the output of the ALU and memory have the most significant impacts on the signal. We believe this difference is mainly due to the different physical parameters of transistors and/or lengths of the connecting wires.

Using this observation, to systematically calculate the activity factors, we use a *linear regression* model:

$$\alpha = \delta + \mathscr{T} \times c + \epsilon, \tag{8.8}$$

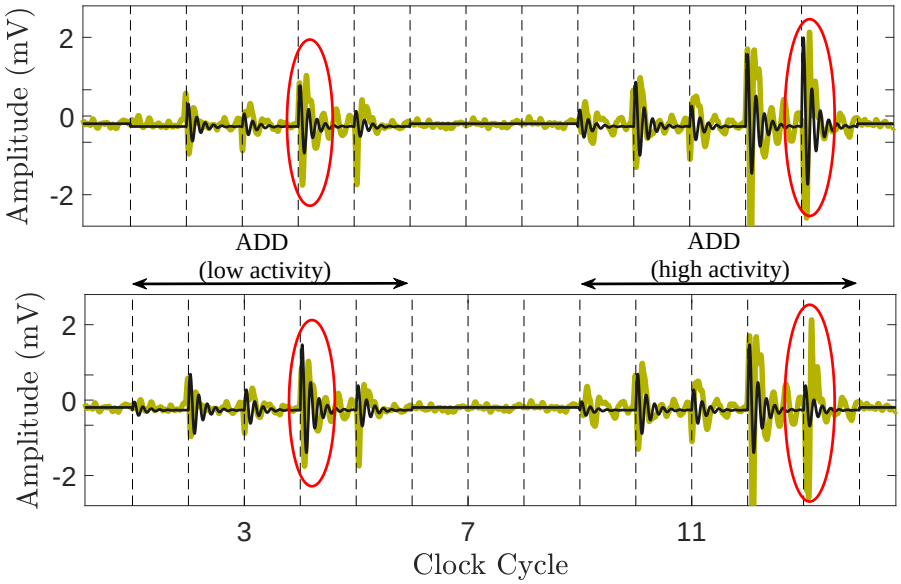


Figure 8.3: Effect of the *activity factor* on the amplitude. The actual signal shown in green. The simulation is shown in black when activity factor is modeled using a linear regression model (top) and when an *average* activity is used (bottom).

where $\mathscr{T}$ is a vector of transition bits across all the existing registers in the targeted pipeline stage, δ and ϵ are the vector of scalar intercept and error terms respectively, and c is the vector of activity factors to be predicted by the model. As mentioned before, α is the scaling factor for the baseline amplitude, A, thus $\alpha = A_{meas}/A_{simul}$. Note that to find $\mathscr{T}$, a detailed micro-architecture model is needed to track all the bit-flips for every gate in the processor (except cache/memory). However, to significantly reduce the complexity and simulation time, the size of $\mathscr{T}$ can be reduced using the step-wise regression method [230] where, iteratively, the size of the fitted model (i.e., α and $\mathscr{T}$ in our case) is reduced using standard statistical metrics such as F-tests [230]. In other words, since *not all the bit-flips have statistically significant impact on the emanated signal*, the non-contributing factors can/should be removed from the model. In our processor, using this method we managed to reduce the size of $\mathscr{T}$ by more than 65%.

Figure 8.3 (top) shows the simulated signals when the linear regression (LR) model is used for activity factors. Compared to the averaging method (bottom), using LR has

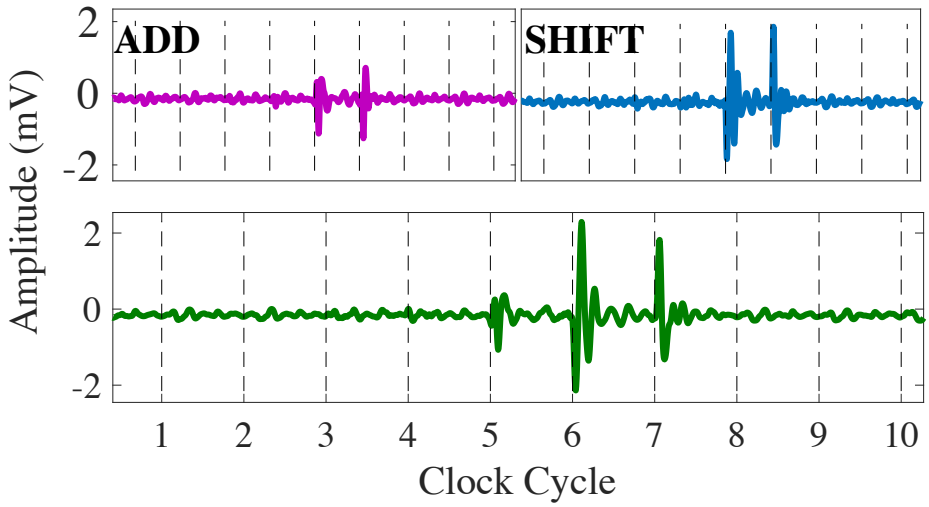


Figure 8.4: An example of how individual sources (pipeline stages) are combined to form the final signal. ***Top***: how the actual EM signal looks like when the instructions are executed in isolation (`NOP, inst, NOP`). ***Bottom:*** The actual EM signal when the instruction sequence is `NOP, ADD, SHIFT, NOP` (i.e., a combination of multiple instruction in the pipeline).

significantly improved the simulation accuracy.

8.4.3 Multi-Input Modeling

Once the signal amplitude for individual sources are calculated, the next step is to combine the signals generated by these individual sources to create the simulated EM signal. In principle, the generated EM signal is the *superposition* of individual waves thus depending on each source's phase, the superposition of each pair can be either *constructive* or *destructive*. Using this fact, the signal can be approximated as a *linear* combination of these individual sources where the coefficients may vary between ± 1, depending on the phases.

Due to the complex nature of the generated EM signals, accurately modeling each and every source mathematically is significantly time-consuming and often infeasible in practice. To tackle this problem and find coefficients for each source, a *model-fitting* approach can be used. We use a *linear-regression* model to find (predict) the overall EM signal. Specifically, we use:

$$X = \delta_s + (\alpha A) \times M + \epsilon_s, \tag{8.9}$$

where αA is the vector of individual sources amplitudes (α is the activity factor and A is the baseline amplitude), δ and ϵ are the intercept and error vectors, M is the predicted coefficients, and X is the final amplitude which will be used in Equ. 8.6.

Figure 8.4 shows an example of how two individual sources are combined in each cycle to form the final signal. Figure 8.4 (top) shows `ADD` and `SHIFT` instructions when they are executed in isolation (i.e., `NOP, inst, NOP`), and Figure 8.4 (bottom) shows how the final signal looks like when the executed sequence is `NOP, ADD, SHIFT, NOP`. Specifically, cycle 6 is when the `ADD` instruction is in `WB` stage and `SHIFT` is in `MEM`, and the resulting signal is a linear combination of these two sources. Note that to find M, we need to measure all the possible combinations of the entire instructions in the ISA, however, as we will show in Section 5, the number of required measurements can be significantly reduced using standard *clustering* algorithms.

8.5 Modeling Micro-Architectural Events

The last step of simulating an EM side-channel signal is adding the signatures of different micro-architectural events to the signal. We particularly add the signatures of the following three events to the signal:

Pipeline Stall. Stalling is a common event in a processor which prevents successor instructions from advancing in the pipeline and preserves the instruction and operands in the stalled stages. Due to this preservation no bit-flips occur in the stalled stages. In addition, to save power, a control signal is typically used to disable (e.g., through power-gating) hardware components in the stalled stages. As a result, stalling typically has a dramatic impact on the switching activities of the stalled stages and, consequently, results in a significant reduction in the amplitude of the generated side-channel signals. Figure 8.5 shows the effect of stalling on the signal where a `MUL` instruction has stalled the pipeline for eight cycles (we intentionally increased the stall cycles in `MUL` for clarity). As can be seen from

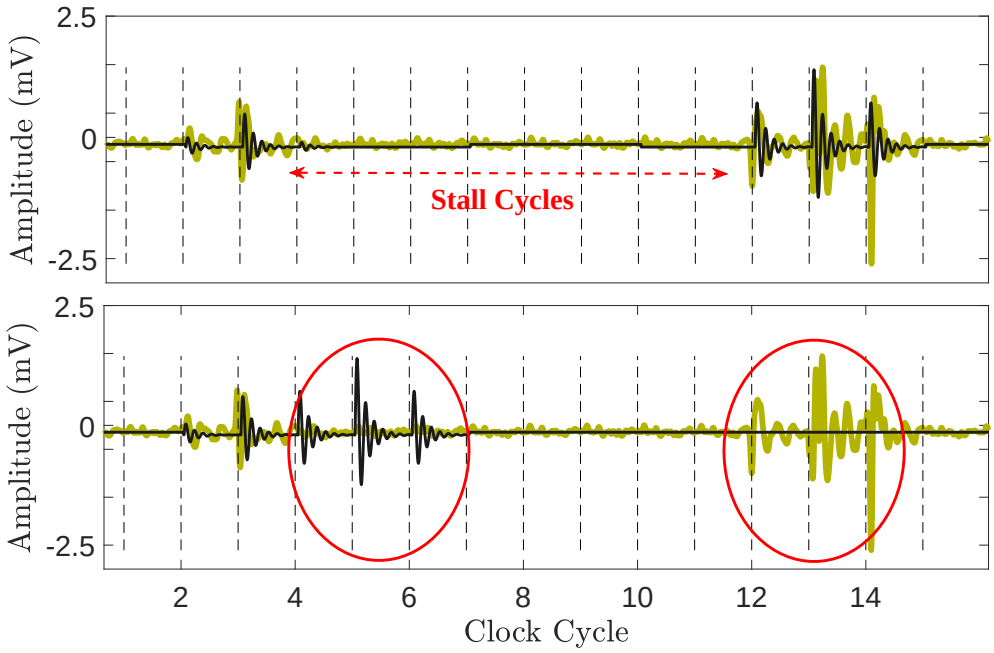


Figure 8.5: Effect of stalls on the signal. The actual signal shown in green, while simulated signals are shown in black when modeling pipeline stalls (top) and not modeling it (bottom).

the figure, not properly simulating stalls (bottom) results in a significant deviation from the original signal (shown in light green).

Note that stalling does not have any impact on prior instructions thus they still generate side-channel signals as they advance through the pipeline. As a result, during stall the received side-channel signal is only generated by the instructions in the non-stalled stages (if any). To properly model stalling, the simulator should be able to detect when stalls are happening (using the micro-architecture model), and ignores the signals generated by the stalled stages during the stall phase. In our model, this is done by setting the amplitudes of stalled stages to zero in Equ. 8.9.

Cache miss. Similar to pipeline stalls, due to a data-dependency, cache miss can also cause stalls. In our design, accessing the cache stalls the pipeline for one cycle. Further, cache miss and access to the memory causes extra two stall cycles. These two signals and their

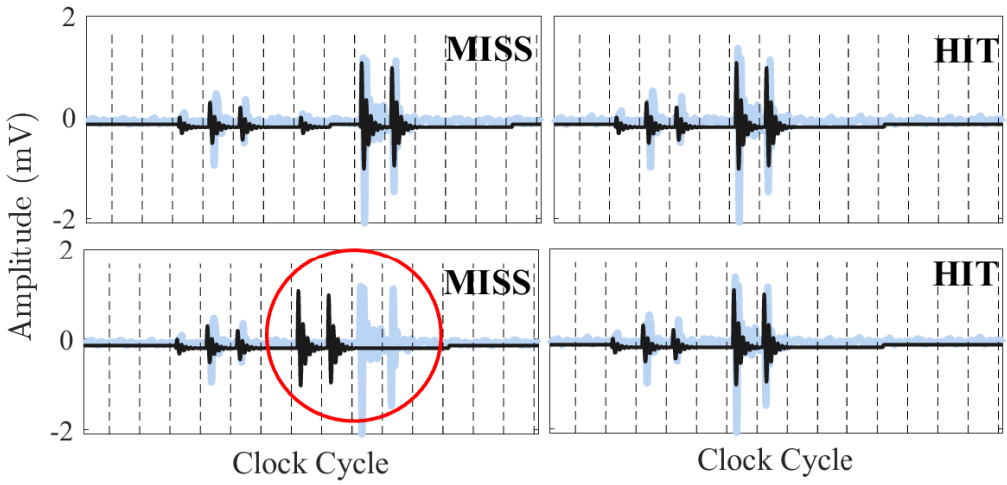


Figure 8.6: Effect of cache-miss (left) and cache-hit (right) on the signal. Miss causes two extra stall cycles. The actual signal (light blue) and simulated signals (black) with (top) and without (bottom) modeling cache misses are shown.

differences are shown in Figure 8.6. As can be seen in the figure, two extra stall cycles (total of three) can be seen in `LD` instruction. Similar to stalls, the cache activity should be properly simulated using the micro-architecture model. Figure 8.6 illustrates how without properly modeling the cache misses the simulated signal will be deviated from the original signal (bottom left).

Misprediction. In addition to stalls, we observed that branch misprediction also has noticeable impact on the side-channel signals. Depending on the pipeline design, the correct outcome of a branch instruction can be resolved after a few cycles (2 cycles in our design), and if a *misprediction* is detected, the processor has to *flush* the incorrectly fetched instructions, and begin executing the correct ones after that. In order to do that, processors typically substitute the incorrect instructions with `NOP` instructions. It is expected that executing these *bubble* instructions temporarily changes the side-channel signals since they change the switching activities of each stage. Figure 8.7 shows the received EM signals with and without a misprediction along with instructions present at each cycle in each pipeline stage.

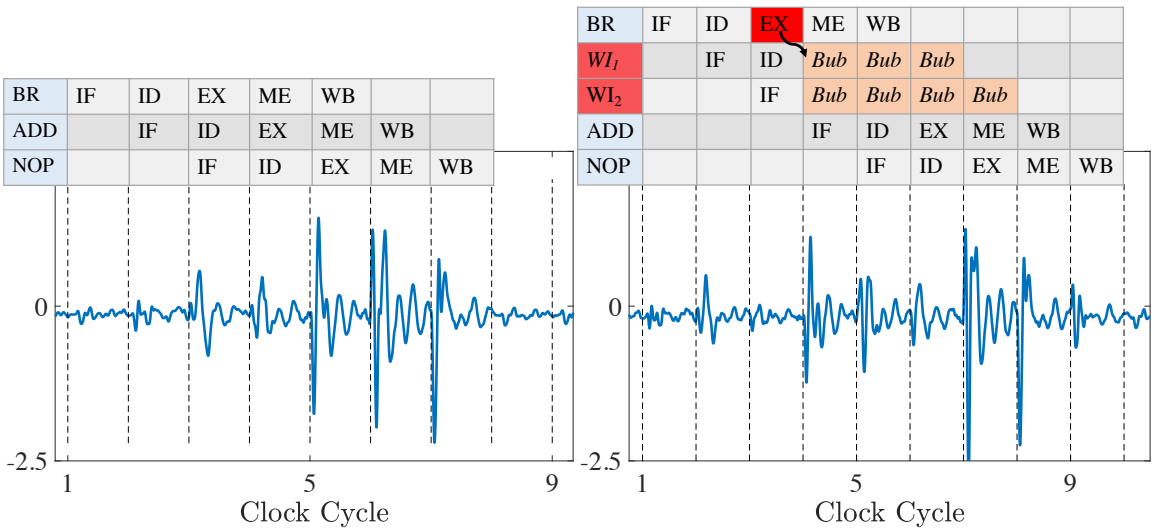


Figure 8.7: Effect of misprediction (right) on the signal. It causes two instructions being flushed from the pipeline and hence affect the signal in those cycles.

Similar to pipeline-stall, using the micro-architecture model, mispredictions can be detected and simulated in our simulator. It is important to mention that we also studied the impact of using different branch-predictors on the side-channel signals (e.g., always not-taken, 2-level, g-share, etc.) and we did not observe any statistically significant difference between these predictors mainly because they have relatively small switching activities (especially for low-end processors).

It is also important to mention that *we tested the effect of other micro-architectural events* such as data-forwarding on the signal and did not observe any significant difference in the presence and/or absence of them. Also note that, as we mentioned before, in this paper, we limited the modeling to bare-metal, and left system-level activities modeling such as interrupts, exceptions, context-switch, etc., and advanced power-saving methods (e.g., DVFS, power gating, etc.) to future work.

8.6 Evaluations

We divide our evaluations into two main parts. First to show the correctness, accuracy, and robustness of our simulator, we present our experimental evaluations on how well

Cluster	Type	Inst.	No. Inst.
1	ALU	`ADD,XOR,JAL, ...`	13
2	Shift	`SLLI,SRT, SRA, ...`	10
3	MUL/DIV	`MUL, DIV, REM, ...`	8
4	Load	`LB, LW, LH, ...`	5
5	Store	`SB, SH, SW`	3
6	Cache	`LB, LW, LH, ...`	5
7	Branch	`BEQ, BLT, BGE, ...`	6

Table 8.1: RISC-V (R32IM) instruction-set and their cluster used in this paper.

the simulated signal *matched* with the original side-channel signal generated by the target hardware for *ALL* possible combinations of the instructions. We then explore the impact of variations such as manufacturing, environmental, etc. on the accuracy of *EMSim*.

The second part of our evaluations (to be presented in §7.7) is focused on the *EMSim* use-cases and its application in different domains such as security, debugging, etc.

8.6.1 Evaluating Model Accuracy

Setup. We implemented a RISC-V based processor on a Terrasic DE0-CV board with an Altera Cyclone-V FPGA [231] with 50 MHz clock-rate. To record side-channel signals, we used a Keysight digital oscilloscope (DSOS804A), with 1 GHz bandwidth and 10 GSa/s rate. We further studied the effect of changing the sampling-rate on the accuracy and found that similar accuracy can be achieved with much lower sampling-rate (about 200 MSa/s in our measurements). As a result, similar results can be achieved using a less expensive device (e.g., TBS1032B Tektronix Digital Oscilloscope [232] costs around $300) and/or a high sampling-rate device can be used for modeling devices with faster clock-rates. To receive EM signals, we used a magnetic probe [89], placed 5 cm above the FPGA. Signal processing is done in Matlab2017-b and the simulator is implemented in standard C++ programming language.

Model Building. In §7.3, we discussed that in order to fit a model, *ALL* possible combi-

nations of instructions should be measured (i.e., about three hundred million combinations in RISC-V ISA). Clearly such a requirement makes the model building extremely time-consuming in practice. However, intuitively, we expect instructions with similar behaviors (e.g., ALU-type, memory-type, etc.) have similar side-channel signals since they share identical hardware activities. Using this intuition, we used the *hierarchical agglomerative algorithm* [233] with the *cross-correlation* as the distance metric to cluster instructions with similar EM pattern into a same cluster. We found that RISC-V ISA can be *clustered* into 7 categories (when the operands are similar) where a single instruction in each category can be a representative of all instructions in that category.

These categories are shown in Table 8.1. Using this table, we then used only a *representative* instruction of the cluster for model building which, in turn, reduce the model building complexity significantly. In our setup, the number of measurements was reduced from 300 million to only 16 thousands. Note that while the clustering algorithm did not use the micro-architecture model as a prior knowledge, the clusters confirmed that instructions with similar micro-architecture activities should be clustered in a same group.

Metric. To measure how well the simulated signals "match" with the real signals, we leverage *normalized cross-correlation* as our metric. To compute that, we first normalize both signals, real and simulated, to have similar average. We then divide each signal to individual clock cycles, and then compare each cycle (between the simulated and the real signals) using cross-correlation as the distance metric. We then define *accuracy* as the average of this cross-correlation across all cycles for all measurements (i.e., we were able to match the waveform in this degree across all possible instruction sequences). Note that we specifically used this approach to show how well the time-domain signal *matches* with the original signal instead of relying on a specific *leakage metric* such as Hamming weight. However, to show the usefulness and versatility of our tool, those results will be shown in the next section.

Benchmark. To prove that our approach provides accurate simulated signals for *ALL* possible instruction combinations thus can be applicable to *ANY* complex program that uses the mixture of the implemented ISA (R32IM), we created a microbenchmark using all possible combinations of the representative instructions shown in Table 8.1. Particularly, for a 5-stage pipeline and 7 distinct clusters, there are $7^5 = 16807$ possible combinations that can appear together (in the pipeline) in a cycle. We created a program to generate all these combinations with random operands. We then manually modified branch instructions and assigned the target address and branch condition to create loops with random instruction and iteration sizes. To limit the execution time, we then randomly put these instructions into groups of 1024 combinations (i.e., 5120 instructions in each group which were executed one after another similar to a real program). To cover all the combinations, 17 of such groups were needed (no two groups were similar). We then executed these randomly-generated groups on the processor normally, and recorded the real and simulated signals. To further prove the validity and correctness of our simulator, we also randomly created another 17 groups, this time from all instructions in the ISA and not just the representatives.

Results. Using these 34 groups/applications described above, we then compared the simulated signals with the actual ones using the our metric defined earlier. Each group/application takes about 9000 cycles to finish on average. The execution-time varied depending on the instructions used and microarchitectural events.

Figure 8.8 shows the simulated and actual EM-side-channel signals for one of the groups tested in our evaluation (for clarity, only the first 50 cycles are shown in the figure). As can be seen from the figure, the simulated signal matches the real signal with high accuracy. We found that, on average, ***EMSim has about 94.1% accuracy in simulating side-channel signals across all possible instruction combinations***.

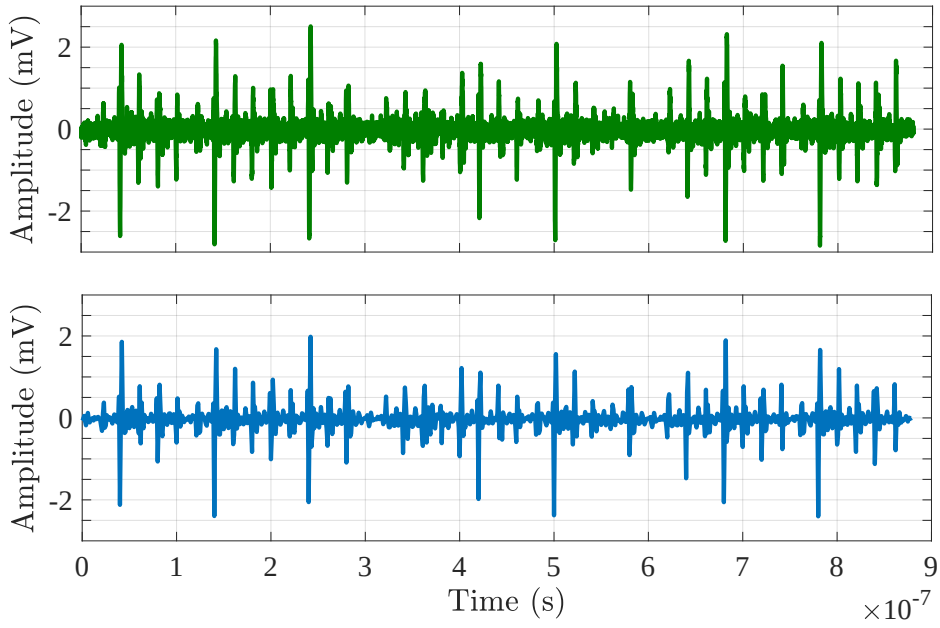


Figure 8.8: A comparison between the signal generated by a real hardware (top) and the simulated signal (bottom) in EMSim.

8.6.2 Manufacturing Variability

To investigate the impact of manufacturing variability on the model's accuracy, i.e., to determine if training is needed for each physical instance of a (same) device, we repeated our measurements for three physical instances of the Terrasic development board. We then compared the signals received form each device using the normalized correlation method. We observed that, for each cycle, the signals for board #2 and #3 are slightly shifted (compared to the board #1), mainly due to the slight shift in the actual clock frequency of the boards. We found that such a shift has no statistically significant impact on the accuracy.

In general, it is important to mention that while a shift in the clock frequency could cause the side-channel signal to be *scaled* in each cycle, we observed that this scaling has (almost) the same impact (in terms of shape and amplitude) on all the instructions thus, effectively, makes the true mutual information unchanged.

8.6.3 Board Variability

To study the effect of board variability, mainly the effect of CMOS technology and board design on the signals, and further evaluate the model accuracy on boards with different manufacturing conditions, we used two other boards: a Terrasic DE1 board with an Altera Cyclone-II FPGA [234] and a Digilent ARTY board with a Xilinx Artix-35T FPGA [235] (both clocked at 50MHz). We then repeated the measurements described in §8.6.1 for these two new boards and found that, using the same processor design and applications, to correctly model the signal both A and c parameters (cf. Equ. 8.9 in §8.4) should be retrained to achieve the similar accuracy. Other parameters such as M remained the same since the position of the antenna and the physical and logic design of the processor stays the same. We envision that for different designs, the baseline amplitude and activity factors should be re-trained (only once) and then can be included by the developers as a library (similar to that of for other properties such as power, timing, etc.).

8.6.4 Effects of Distance

Transferring the ideas from communication theory literature, to find the effect of the distance (i.e., the position of the probe and its distance to the center of board) on each source, a parameter, called ***loss-coefficient*** or β, can be considered as the channel coefficient of a flat-fading channel. Here, we need to note that, regardless of the position of the probe, the baseline amplitude, A, can not be measured solely because we do not have any control on the power distribution of the board for each instruction at each pipeline stage. Hence, *the resulting signal power is always a combination of the actual signal amplitude and the corresponding loss coefficient (i.e., $A\beta$).* However, to deal with this problem, we choose the probe's location at the center of the processor as the *base point*, and define β_0 as the loss coefficient at this point. Further denoting A_0 is the actual emanated signal amplitude, we assume the amplitude of the signal can be written as $A = A_0\beta_0$ with respect to the base point. Therefore, with these assumptions, β is assumed to be one for all the measurements

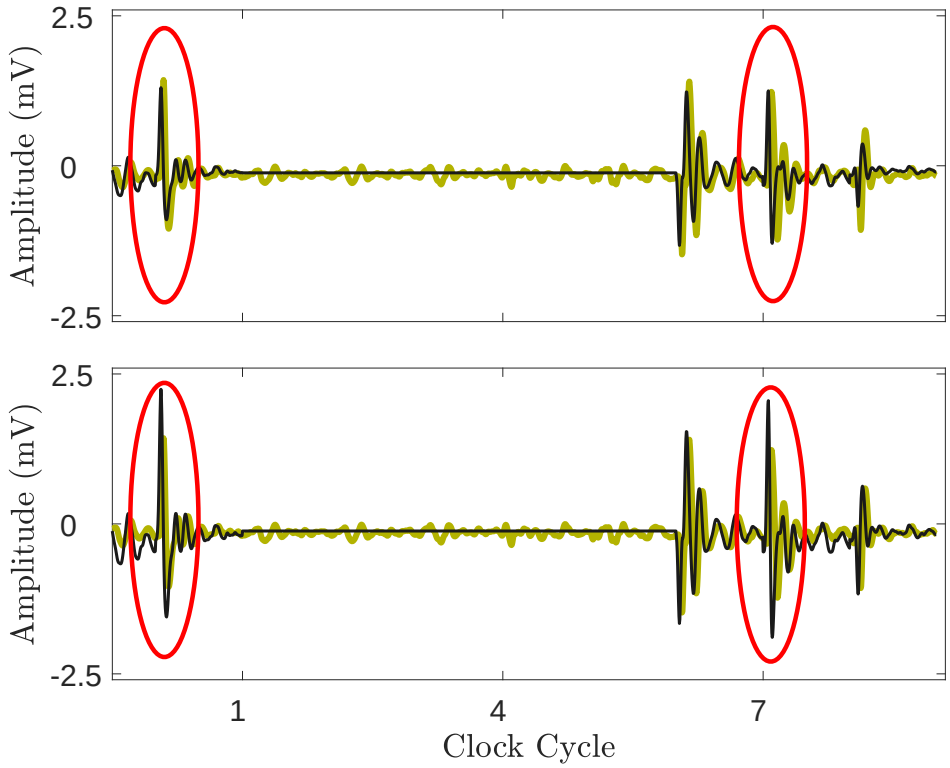


Figure 8.9: Effect of distance on the signal amplitude. For both figures, the plots with darker color correspond to reconstructed signal, and the other ones correspond to the original signal.

done in this paper.

To further investigate the effect of β on the amplitude, we measured the signal (with the same input trace) at a different location, and compared the results with the base case. Figure 8.9 illustrates the effect of the antenna location on the loss coefficient factor β. Here, the training signals for the reconstruction are obtained from the base measurement, and the figure at the bottom is obtained by neglecting the effect of β (i.e., $\beta = 1$) during the simulation. The figure at the top is generated by solving the same linear regression model given in Equ. 8.9, this time by substituting A by $A\beta$, where β is not constrained to one any more (while A is the signal obtained in the base case). We can conclude that considering the effect of β is crucial to explain the changes due to antenna location since better correlation and root mean square results are obtained with the adjusted β. Note that, adjusting the β is only required during the model building (i.e., during measurements if the position of

the probe changes), however, the user of the tool does not require to change/adjust β for his/her leakage estimation and can use the base case or numerous cases (depending on the availability) to obtain an "average" leakage estimation, or "worst" case for $\beta = 1$.

8.7 Practical Use-Cases for EMSim

This section describes several example use cases for *EMSim*'s ability to accurately simulate side-channel signals.

8.7.1 Side-Channel Leakage Estimation

An important step for defending against side-channel attacks (SCA) is estimating how much (sensitive) information can be possibly leaked (through a specific or set of side-channels) during the execution of an application. To estimate this leakage, different metrics can be used. Particularly, for EM side-channels the state-of-the-art methods are Test Vector Leakage Assessment (TVLA) [236, 237] and Signal Available to Attacker (SAVAT) [36] methodologies.

Due to the lack of simulation tools, to properly calculate these metrics, several actual measurements should be performed. Unfortunately, these measurements often require sophisticated equipment and experts with various skills which, in turn, makes them expensive and difficult in practice. Using our approach, however, we show that *EMSim* is capable of generating highly-accurate simulated signals which can be used to calculate these metrics precisely which eliminates the need for an actual measurement infrastructure.

The following describes how *EMSim* can be used to model TVLA and SAVAT. It is important to mention that unlike prior work [221, 222, 238, 239], *EMSim is NOT limited to a specific metric or analysis, and it can be used for ANY analysis based on the EM signal.*

Test Vector Leakage Assessment (TVLA). This metric is based on *T-test*, a statistical test which determines if there is a significant difference between the *means* of two groups. In

	LDM		LDC		NOP		ADD		MUL		DIV	
	R	S	R	S	R	S	R	S	R	S	R	S
LDM	0.02	0	3.71	3.91	5.34	5.32	5.24	5.20	5	5.02	4.98	4.98
LDC	3.72	3.91	0.04	0	0.81	0.85	0.74	0.74	0.21	0.24	0.21	0.23
NOP	5.35	5.32	0.8	0.86	0.01	0	0.08	0.1	0.67	0.69	0.66	0.69
ADD	5.24	5.20	0.74	0.75	0.07	0.1	0.03	0	0.98	1.05	1.03	1.1
MUL	4.98	5.01	0.22	0.21	0.66	0.68	0.94	1	0.03	0	0.04	0.01
DIV	4.97	4.99	0.21	0.21	0.65	0.68	1.05	1.13	0.03	0.01	0.02	0

Table 8.2: Signal Available to Attacker metric [36] for Real measurements (R) and Simulations (S).

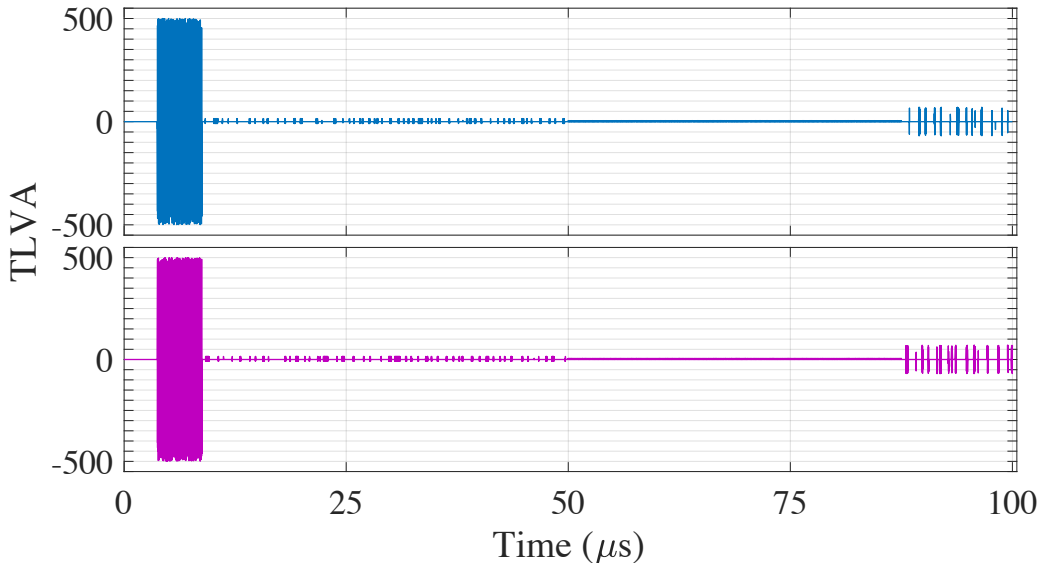


Figure 8.10: AES-128 leakage assessment using TLVA methodology on the measured/actual signal (top) and the simulated signal (bottom).

the context of SCA, TLVA shows how much the side-channel signal (e.g., EM) is correlated with specific values in the code. The values can be either some known intermediate nodes (e.g., the output of *Sbox* in AES) or more generally, some *fixed* (or *semi-fixed*) input parameters of a specific function. If the metric exceeds a threshold (i.e., a confidence value in the statistical test), it indicates that there is a (potential) leakage for an SCA such as DPA [240, 241], template attacks [198], etc.

To compare the accuracy of TLVA metric based on the measured vs. simulated signals, we ran AES-128 on our RISC-V processor. We then used our setup described in §8.6.1, to

measure the signal. Figure 8.10 shows these measurements for the real signal (top) and the simulated signal (bottom). As can be seen from the figure, the TLVA metric based on the simulated signal is highly matched with the real measurement and follows the same pattern (and values) as the actual one (i.e., *no-activity*→*high*→*low*→*no*→*medium* sequence).

Signal Available to Attacker (SAVAT). This metric measures the side-channel signal created by a specific single-instruction difference in program execution, i.e., the amount of signal made available to an attacker who wishes to decide whether the program has executed instruction/event *A* or instruction/event *B*.

To measure this metric, Callan *et al.* [36] developed a microbenchmark which creates a controlled alternation between A and B instructions many times. Such alternation creates a periodic signal with period $t_p = t_A + t_B$, where for the half of the period *A* is executing and for the other half *B*. Such a periodic activity can then be observed in the frequency domain as a spike at $f_p = 1/t_p$. The key insight is that the corresponding energy of the spike (i.e., area under the curve) indicates how different *A* and *B* are from each other (in terms of side-channel signals) hence reveals how much signal would be available to an attacker when the difference between two samples is whether *A* was executed or *B*.

To compute SAVAT in both real measurements and simulated signals, we implemented the microbenchmark proposed by Callan *et al.* [36], and used the setup explained in §8.6.1 to collect the signals. Table 8.2 shows SAVAT values in our processor for 6 pairs of instructions. As can be seen, the values retrieved from simulations are highly matched with the values computed using the real measurements. SAVAT can then be utilized to reveal the information leakage capacity of the system [242].

8.7.2 Application to Debugging/Profiling

While so far we have shown how EMSim can be utilized to accurately model EM side-channel signals and thus can be used for leakage estimation during developing secure soft-

ware, in this section, we present another potentially useful use-case of EMSim and show how hardware designers and computer architects can also leverage this framework during hardware development.

Given that EMSim can accurately model the system for each pipeline stage and each micro-architecture event, it can potentially be used as a debugging tool in the chip-design flow such as a debugging tool for finding design bugs in post-place-and-route stage and/or for finding manufacturing bugs/defects in post-fabrication. In contrary with signal modeling, in this scenario, the signals simulated by the simulator can be assumed as the "ground-truth" or "expected" signal where the signals emanated by the hardware have to be matched to these *reference* models. A deviation from the *reference* model obtained by the simulations indicates that there is an unwanted change/error in the hardware.

The main advantage of this approach compared to existing standard testing methods is that the proposed approach is ***zero-overhead*** and does not require any testing infrastructure on the system which, in turn, saves a significant amount of area and reduces complexity.

To further demonstrate the feasibility of this approach, Figure 8.11 shows a scenario where there is a bug in designing a multiplier in the Execution stage. The multiplier is designed such that it calculates the result of multiplying two 16-bits operands in three cycles where the majority of the activity (i.e., writing the output register, etc.) takes place in the last (third) cycle. However, as seen from the figure, the amplitude of the measured signal (top) in the third cycle of the execution (shown in a red circle) is significantly lower than that of in the simulation (bottom). Further investigation reveals that instead of properly multiplying two 16-bits data, the designed multiplier only uses the lower half (i.e., 8-bit data) of each operand and ignores the upper half of those inputs hence results in a significantly lower activity factor and thus much smaller signal strength.

It is important to mention that this method, fundamentally, relies on the signal detection granularity (i.e., how fine-grained changes can be detected), thus it may not be useful in cases where the change in the signal is smaller than the algorithm's detection granularity.

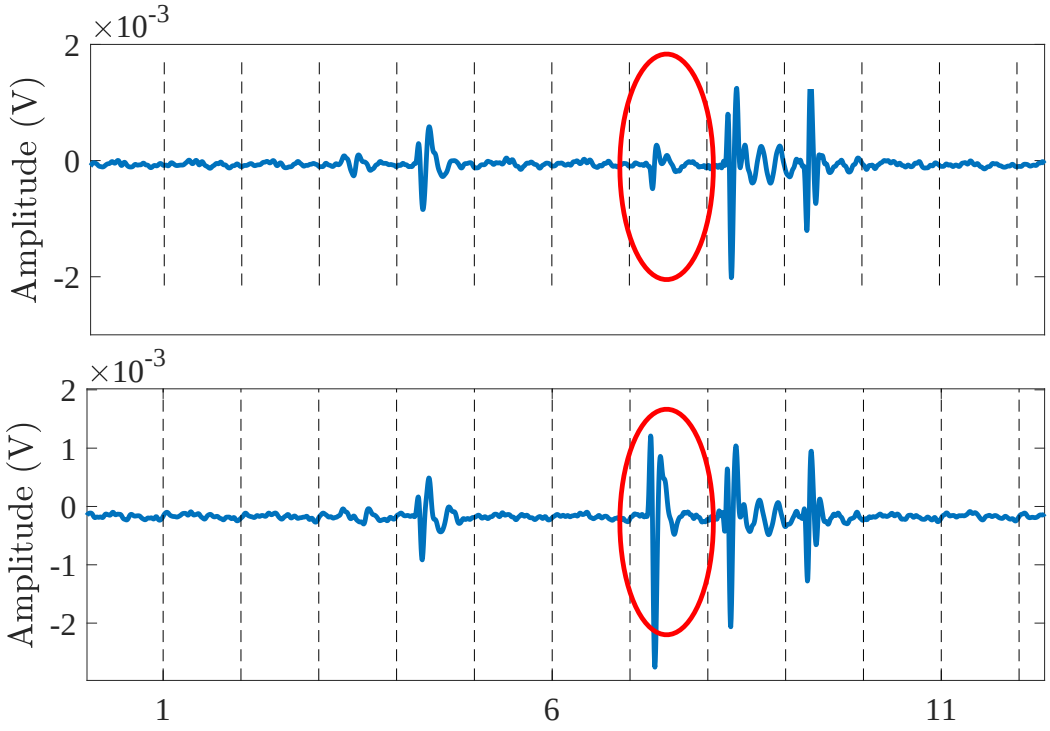


Figure 8.11: A case-study to show how EMSim can be used for debugging. The measured signal (top) does not match with the reference model obtained by the simulation model (bottom) which indicates that there is a potential error/issue in the hardware.

8.8 Prior Work

Much work has been done to characterize particular analog side-channels [243, 244, 245, 223] and prevent attacks that use them [49, 246, 217, 39, 47, 247, 248], either by removing the tie between sensitive information and the side-channel signal, or by trying to make the signal more difficult to measure. However, such work mostly focuses on preventing a particular attack in a specific piece of code and are less focused about the relationship between the hardware, software, and the side-channel signal.

Strategies for quantifying potential side channel exposure at the micro-architectural and architectural levels are still an open problem. Existing work proposed different methods and/or metrics to estimate the leakage either for a specific type of side-channel (e.g., cache, power, EM, etc.) or alternatively, as a generic framework to estimate the overall leakage for any given side-channel.

Side-Channel Vulnerability Factor (SVF) [217] measures how the side-channel signal correlates with high-level execution patterns (e.g., program phase transitions). While this metric allows overall assessment of the "leakiness" of a particular system and application over a given side-channel, it provides limited insight to 1) computer architects about which architectural and microarchitectural features are the strongest leakers, and to 2) software developers about how to reduce the side-channel leakiness of their code.

To address these limitations, the SAVAT (Signal Available to Attacker) metric was proposed, along with practical methods to measure SAVAT in practice [36, 218, 219, 220]. SAVAT measures the side-channel signal (particularly EM and power from laptops) created by a specific single-instruction difference in program execution, i.e., the amount of signal made available to a potential attacker who wishes to decide whether the program has executed instruction/event A or instruction/event B. These measurements can be used to determine the potential for information leakage when execution of individual instructions or even sections of code depend on sensitive information. Unfortunately, SAVAT only quanti-

fies the overall difference between two specific instructions, but ignores the time at which these differences occur and how they are afected by other instructions in the program.

Similar to SAVAT, McCann *et al.* [221] proposed a modeling technique capable of producing a leakage metric at instruction-level for power (and/or EM) side-channel signals on ARM M0/M4 cores. To estimate the leakage for individual instructions, the proposed method only requires knowledge about different characteristics of the system at ISA-level such as data-dependent effects of neighboring instructions in a sequence, register effects, bit-flips, etc. Like SAVAT, that method provides interesting insights about possible sources of leakage, but ignores the effects of micro-architecture events such as cache miss, branch mis-prediction, etc. on the signal. As we show in this paper, that may lead to making wrong conclusions about the leakage model of the software/system.

Very recently, Gorman *et al.* [249] designed MESC, an architectural framework for modeling electromagnetic emanations from a core. MESC takes into account the layout and the switching activity of a process, and accounts for the effects of shielding and environmental noise. While this work provides accurate results for different cores with different layout, MESC also provides no information about micro-architecture events which could lead to some non-negligible inaccuracies.

Another related work to EMSim is the method proposed by Barenghi and Pelosi [222] where the leakage for individual instructions was calculated by measuring the power consumption between two consecutive cycles and employing the Pearson correlation coefficient between the two measurements. To calculate the leakage, in addition to leveraging the ISA-level information, pipeline model was also used. However, the framework did not consider any micro-architecture events, nor pipeline stalls and only accounts the number of cycles that takes for each instruction to execute. It also did not model the individual effect of each stage on the others and the overall signal. In this work, however, we took a more systematic and accurate approach by considering different micro-architecture events and hardware effects.

Also related to this work are work on leveraging EM signals for profiling [183, 127, 250, 213, 186, 185, 129]. EMPROF [250] analyzes the system's EM emanations to identify processor stalls that are associated with last-level cache (LLC) misses. Compared to these frameworks, instead of leveraging the existing EM side-channel signal for profiling, our work takes a more holistic approach and systematically models the underlying relation between the software and hardware and provides insights on how and why these EM side-channel signals are created due to a variety of micro-architectural and hardware activities. Using this approach, EMSim can be used for a variety of purposes beyond only for program/memory profiling (e.g., hardware modeling, leakage estimation, compiler development, etc.).

Another body of work related to this paper are the cycle-accurate models/tools to simulate power and/or microarchitecture [200, 201, 202, 203, 147, 92, 204, 205]. While these models can accurately model the power consumption at each cycle, they are different from this work and hence may not be a proper tool for simulating EM side-channel signals for two main reasons: *First*, while these methods do consider the activity factor to calculate power, they often treat all the bit-flips equally. However, as shown in this paper, depending on the design, not all flips equally contribute to the overall signal. Ignoring this fact can lead to inaccurate modeling (see Figure 8.3). *Second*, depending on the architecture, different stages might have different effect on each other and the overall signal. Without properly modeling these effects, the overall signal can not be modeled (see Figure 8.9).

More indirectly related to our work are methods for instruction tracking/intrusion detection based on side-channel signals [251, 105, 252, 253]. These methods often use different signal processing methods (e.g., Markov Model, Statistical tests, etc.) and/or machine learning techniques to find the most likely executed instruction based on the side-channel trace. While they are effective in providing a non-intrusive, zero-overhead method for profiling/intrusion detection, they are not designed to model the signal and/or provide any information or insight about how these signals are generated.

8.9 Conclusions

This Chapter presented *EMSim*, an approach that enables simulation of the EM side-channel signals cycle-by-cycle using a detailed micro-architectural model of the device. Our evaluation of *EMSim* finds that its simulation-derived signals closely matches signals measured from real hardware. To gain further insight, we also experimentally identified how the accuracy of the simulated signals degrades when key micro-architectural features and other hardware behaviors are omitted from the simulation model.

We envision a variety of uses for EMSim. For hardware, software, and compiler developers, it allows EM leakage to be quantified without having to build actual hardware and/or actually measure signals. More importantly, it allows simulated signals to be broken down and attributed to specific parts of the hardware and software. Furthermore, when hardware prototypes are available, significant discrepancies between the signal generated by EMSim and actual EM emanations can be used to identify where the actual design differs from the simulated microachitecture, which can be used to debug the hardware and/or to refine the simulation model to more closely match the hardware.

We believe that *EMSim* can be extended to more complex processors by using a similar multi-input-single-output methodology, where each pipeline stage acts as a single source. For out-of-order processors, we expect higher *baseline hardware amplitude* for each stage as the hardware of each stage becomes more complex. We also expect different values for activity factors and coefficients for individual stages. Further, since an OoO processor has more *shared* units, to accurately model the signal, these shared units should be carefully simulated and their signals added to each cycle. Nonetheless, since the root cause of creating side-channel signals are bit-flips at the gate-level, we do not expect any fundamental *modeling* difference between in-order and OoO designs.

CHAPTER 9
SUMMARY AND FUTURE WORK

9.1 Summary

This thesis investigated two important research questions regarding to the security of embedded systems and the role of side-channels in their security.

The first question was *"Can (and how) side-channel signals be used for useful purposes?"* To answer this question, we developed a novel method called *Spectral Profiling* which leveraged analog-domain side-channels for monitoring a resource-constrained system. The main idea behind this method was we can simplify signal analysis by knowing this fact that most programs in embedded system domain tend to have repetitive phases. We then developed a novel signal processing methodology to effectively monitor embedded system with high accuracy.

Further, we showed how Spectral Profiling can be used in a variety of applications including profiling, intrusion detection, and establishing trust. We showed this method is quite powerful even in noisy environment. We believe the contributions presented in the first part of this thesis can significantly improve the security of resource-constrained devices where limitations (e.g., lack of infrastructure, overheads, etc.) make them vulnerable against adversaries.

The second question that we investigated was *"how analog side-channel leakages can be mitigated?"* To answer that question, we made two main contributions. We first discovered a new side-channel vulnerability on modern systems and showed how ignoring security as a first-order factor can lead to severe vulnerabilities. Second, to fundamentally address this problem, we developed a novel simulation tool which can estimate the analog side-channel signals for any given software. Such a tool could be used by a developer at the

design-stage of software and hence, can significantly reduce the vulnerability of the system against analog side-channels.

9.2 Future Work

This decade has already seen a significant surge in the number of cyber-attacks. With the exponential growth of computers in numbers and their ever-increasing importance in controlling critical tasks, it is expected that cyber-security and data privacy become even more serious problems in the next decade. To this end, understanding and properly modeling side-channels become even more important tasks.

The largest opportunity for future work is the improvement and extension of the the proposed simulation tool, EMSim. In this thesis we showed how EMSim can be used to estimate side-channel signals for a relatively simple processor. However, such a tool can be extended to cover a variety of systems and architectures. Specifically, it can be extended to model *out-of-order processors*. Moreover, such a tool can even be extended beyond the processor, and can model other important components in the system including the main memory, I/O, etc., to provide a much accurate model about the system's behavior.

Another possible future direction is extending the idea of Spectral Profiling to more complex systems and applications. For example, an important extension to our method is the ability to monitor multi-thread/multi-core systems where several application concurrently create side-channel signals. The critical step in creating such a framework is adding a pre-proccesing unit which analyzes the input signal and successfully *separate* it into multiple sources/cores (e.g., by using standard blind source separation techniques such as ICA). The separated signal can then be fed into an existing framework (e.g., EDDIE, REMOTE, etc.) for further analysis (e.g., malware detection).

REFERENCES

[1] Intel, *A guide to the internet of things infographic*, https://www.intel.com/content/www/us/en/internet-of-things/infographics/guide-to-iot.html, 2017 (accessed Feb. 2, 2018).

[2] M. L. Michael Chui and R. Roberts, "The internet of things," *McKinsey Quarterly*, Mar. 2010.

[3] internet-of More-Things.com, *Security is essential for the growing iot*, https://internetofmorethings.com/iot-security-infographic/, 2017 (accessed Feb. 1, 2018).

[4] I. Zeifman, D. Bekerman, and B. Herzberg, *Source code for iot botnet mirai released*, https://krebsonsecurity.com/2016/10/source-code-for-iot-botnet-mirai-released, 2016 (accessed Oct. 10, 2017).

[5] Z.-K. Zhang, M. C. Y. Cho, C.-W. Wang, C.-W. Hsu, C.-K. Chen, and S. Shieh, "Iot security: Ongoing challenges and research opportunities," in *Proceedings of the 2014 IEEE 7th International Conference on Service-Oriented Computing and Applications*, ser. SOCA '14, Washington, DC, USA: IEEE Computer Society, 2014, pp. 230–234, ISBN: 978-1-4799-6833-6.

[6] K. Dunham, "Evaluating anti-virus software: Which is best?" *Information Systems Security*, vol. 12, no. 3, pp. 17–28, 2003.

[7] C. Willems, T. Holz, and F. C. Freiling, "Toward automated dynamic malware analysis using cwsandbox," *IEEE Security & Privacy*, vol. 5, no. 2, pp. 32–39, 2007.

[8] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13, Tel-Aviv, Israel: ACM, 2013, pp. 559–570, ISBN: 978-1-4503-2079-5.

[9] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Malware-aware processors: A framework for efficient online malware detection," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 651–661.

[10] M. Ozsoy, K. N. Khasawneh, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Hardware-based malware detection using low-level architectural features," *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3332–3344, 2016.

[11] A. Viswanathan, K. Tan, and C. Neuman, "Deconstructing the assessment of anomaly-based intrusion detectors," in *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 8145*, ser. RAID 2013, Rodney Bay, St. Lucia: Springer-Verlag New York, Inc., 2013, pp. 286–306, ISBN: 978-3-642-41283-7.

[12] B. B. Kang and A. Srivastava, "Dynamic malware analysis," in *Encyclopedia of Cryptography and Security, 2nd Ed.* 2011, pp. 367–368.

[13] J. Demme and S. Sethumadhavan, "Rapid identification of architectural bottlenecks via precise event counting," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11, 2011.

[14] A. Ketterlin and P. Clauss, "Profiling data-dependence to assist parallelization: Framework, scope, and optimization," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45, 2012.

[15] Q. Zhao, I. Cutcutache, and W. Wong, "Pipa: Pipelined profiling and analysis on multi-core systems," in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '08, 2008.

[16] S. Wallace and K. Hazelwood, "Superpin: Parallelizing dynamic instrumentation for real-time performance," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '07, 2007.

[17] X. Yang, S. M. Blackburn, and K. S. McKinley, "Computer performance microscopy with shim," in *ACM/IEEE International Symposium on Computer Architecture*, ser. ISCA '15, 2015.

[18] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *23rd USENIX Security Symposium*, 2014, pp. 719–732, ISBN: 978-1-931971-15-7.

[19] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P)*, 2015, pp. 605–622, ISBN: 978-1-4673-6949-7.

[20] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, 2009, pp. 199–212, ISBN: 978-1-60558-894-0.

[21] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?" In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 168–179.

[22] G. Irazoqui, T. Eisenbarth, and B. Sunar, "SA: A shared cache attack that works across cores and defies VM sandboxing and its application to AES," in *2015 IEEE Symposium on Security and Privacy (S&P)*, 2015, pp. 591–604.

[23] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in javascript and their implications," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015, pp. 1406–1418, ISBN: 978-1-4503-3832-5.

[24] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache attacks on mobile devices," in *25th USENIX Security Symposium*, 2016, pp. 549–564, ISBN: 978-1-931971-32-4.

[25] D. Evtyushkin and D. Ponomarev, "Covert channels through random number generator: Mechanisms, capacity estimation and mitigations," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 843–857, ISBN: 978-1-4503-4139-4.

[26] S. K. Khatamifard, L. Wang, S. Köse, and U. R. Karpuzcu, "A new class of covert channels exploiting power management vulnerabilities," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 201–204, 2018.

[27] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-cpu attacks," in *25th USENIX Security Symposium*, 2016, pp. 565–581, ISBN: 978-1-931971-32-4.

[28] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: High-speed covert channel attacks in the cloud," in *21st USENIX Security Symposium*, 2012, pp. 159–173, ISBN: 978-931971-95-9.

[29] M. Alagappan, J. Rajendran, M. Doroslovački, and G. Venkataramani, "DFS covert channels on multi-core platforms," in *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2017, pp. 1–6.

[30] O. Aciiçmez, C. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2007, pp. 312–320, ISBN: 1-59593-574-6.

[31] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *Proceedings of the*

Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2018, pp. 693–707, ISBN: 978-1-4503-4911-6.

[32] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Understanding and mitigating covert channels through branch predictors," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 1, 10:1–10:23, 2016.

[33] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, "Rendered insecure: GPU side channel attacks are practical," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018, pp. 2139–2153, ISBN: 978-1-4503-5693-0.

[34] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *27th USENIX Security Symposium*, 2018, pp. 955–972, ISBN: 978-1-931971-46-1.

[35] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, "The EM side-channel(s)," in *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2003, pp. 29–45, ISBN: 3-540-00409-2.

[36] R. Callan, A. Zajić, and M. Prvulovic, "A practical methodology for measuring the side-channel signal available to the attacker for instruction-level events," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Cambridge, United Kingdom, 2014, pp. 242–254, ISBN: 978-1-4799-6998-2.

[37] D. Genkin, L. Pachmanov, I. Pipman, and E. Tromer, "Stealing keys from PCs using a radio: Cheap electromagnetic attacks on windowed exponentiation," in *Cryptographic Hardware and Embedded Systems (CHES)*, 2015, pp. 207–228, ISBN: 978-3-662-48324-4.

[38] A. Zajić and M. Prvulovic, "Experimental demonstration of electromagnetic information leakage from modern processor-memory systems," *IEEE Transactions on Electromagnetic Compatibility*, vol. 56, no. 4, pp. 885–893, 2014.

[39] Y. Han, S. Etigowni, H. Liu, S. Zonouz, and A. Petropulu, "Watch me, but don't touch me! contactless control flow monitoring via electromagnetic emanations," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, Dallas, Texas, USA: ACM, 2017, pp. 1095–1108, ISBN: 978-1-4503-4946-8.

[40] Z. Hadjilambrou, S. Das, M. A. Antoniades, and Y. Sazeides, "Sensing CPU voltage noise through electromagnetic emanations," *IEEE Comput. Archit. Lett.*, vol. 17, no. 1, pp. 68–71, 2018.

[41] M. Guri, G. Kedma, A. Kachlon, and Y. Elovici, "Airhopper: Bridging the air-gap between isolated networks and mobile phones using radio frequencies," in *9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE)*, 2014, pp. 58–67.

[42] M. Alam, H. A. Khan, M. Dey, N. Sinha, R. Callan, A. Zajic, and M. Prvulovic, "One&done: A single-decryption EM-based attack on OpenSSL's constant-time blinded RSA," in *27th USENIX Security Symposium*, 2018, pp. 585–602, ISBN: 978-1-931971-46-1.

[43] A. G. Bayrak, F. Regazzoni, P. Brisk, F. Standaert, and P. Ienne, "A first step towards automatic application of power analysis countermeasures," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2011, pp. 230–235.

[44] D. Brumley and D. Boneh, "Remote timing attacks are practical," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, 2003, pp. 1–1.

[45] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom, "ECDSA key extraction from mobile devices via nonintrusive physical side channels," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Vienna, Austria, 2016, pp. 1626–1638, ISBN: 978-1-4503-4139-4.

[46] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology (CRYPTO)*, 1999, pp. 388–397, ISBN: 978-3-540-48405-9.

[47] Y. Liu, L. Wei, Z. Zhou, K. Zhang, W. Xu, and Q. Xu, "On code execution tracking via power side-channel," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, Vienna, Austria: ACM, 2016, pp. 1019–1031, ISBN: 978-1-4503-4139-4.

[48] B. Farshteindiker, N. Hasidim, A. Grosz, and Y. Oren, "How to phone home with someone else's phone: Information exfiltration using intentional sound noise on gyroscopic sensors," in *10th USENIX Workshop on Offensive Technologies (WOOT)*, 2016.

[49] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder, "Acoustic side-channel attacks on printers," in *Proceedings of the 19th USENIX Conference on Security*, Washington, DC, 2010, pp. 20–20, ISBN: 888-7-6666-5555-4.

[50] D. Genkin, A. Shamir, and E. Tromer, "RSA key extraction via low-bandwidth acoustic cryptanalysis," in *Advances in Cryptology (CRYPTO)*, 2014, pp. 444–461, ISBN: 978-3-662-44371-2.

[51] T. Zhu, Q. Ma, S. Zhang, and Y. Liu, "Context-free attacks using keyboard acoustic emanations," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer*

and Communications Security (CCS), 2014, pp. 453–464, ISBN: 978-1-4503-2957-6.

[52] C. Song, F. Lin, Z. Ba, K. Ren, C. Zhou, and W. Xu, "My smartphone knows what you print: Exploring smartphone-based side-channel attacks against 3D printers," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 895–907, ISBN: 978-1-4503-4139-4.

[53] T. Halevi and N. Saxena, "Acoustic eavesdropping attacks on constrained wireless device pairing," *IEEE Transactions on Information Forensics and Security*, vol. 8, no. 3, pp. 563–577, 2013.

[54] J. Brouchier, T. Kean, C. Marsh, and D. Naccache, "Temperature attacks," *IEEE Security and Privacy*, vol. 7, no. 2, pp. 79–82, Mar. 2009.

[55] R. J. Masti, D. Rai, A. Ranganathan, C. Müller, L. Thiele, and S. Capkun, "Thermal covert channels on multi-core platforms," in *24th USENIX Security Symposium*, 2015, pp. 865–880, ISBN: 978-1-931971-232.

[56] Z. Long, X. Wang, Y. Jiang, G. Cui, L. Zhang, and T. Mak, "Improving the efficiency of thermal covert channels in multi-/many-core systems," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018, pp. 1459–1464.

[57] D. Genkin, I. Pipman, and E. Tromer, "Get your hands off my laptop: Physical side-channel key-extraction attacks on pcs," in *Proceedings of the 16th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2014, pp. 242–260, ISBN: 978-3-662-44708-6.

[58] Y. Su, D. Genkin, D. Ranasinghe, and Y. Yarom, "USB snooping made easy: Crosstalk leakage attacks on USB hubs," in *Proceedings of the 26th USENIX Conference on Security Symposium*, Vancouver, BC, Canada, 2017, pp. 1145–1161, ISBN: 978-1-931971-40-9.

[59] P. Marquardt, A. Verma, H. Carter, and P. Traynor, "(Sp)iPhone: Decoding vibrations from nearby keyboards using mobile phone accelerometers," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011, pp. 551–562, ISBN: 978-1-4503-0948-6.

[60] R. Ogen, K. Zvi, O. Shwartz, and Y. Oren, "Sensorless, permissionless information exfiltration with Wi-Fi micro-jamming," in *12th USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

[61] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A defense against cache timing attacks in speculative execution processors," in *2018*

51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2018, pp. 974–987.

[62] H. Highland, "Electromagnetic radiation revisited," *Computers and Security*, pp. 85–93, 1986.

[63] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, "The EM side-channel(s)," in *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2002*, 2002, pp. 29–45.

[64] A. Zajić and M. Prvulovic, "Experimental demonstration of electromagnetic information leakage from modern processor-memory systems," *IEEE Transactions on Electromagnetic Compatibility*, vol. 56, no. 4, pp. 885–893, 2014.

[65] J. Young, "How old is Tempest?" *Online response collection, http://cryptome.org/tempest-old.htm*, 2000.

[66] M. G. Khun, "Compromising emanations: eavesdropping risks of computer displays," *The complete unofficial TEMPEST web page: http://www.eskimo.com/˜joelm/tempest.html*, 2003.

[67] W. van Eck, "Electromagnetic radiation from video display units: an eavesdropping risk?" *Computers and Security*, pp. 269–286, Dec. 1985.

[68] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: concrete results," in *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2001*, 2001, pp. 251–261.

[69] H. Sekiguchi and S. Seto, "Measurement of radiated computer rgb signals," *Progress in Electromagnetic Research C*, pp. 1–12, 2009.

[70] Y. Suzuki and Y. Akiyama, "Jamming technique to prevent information leakage caused by unintentional emissions of pc video signals," in *Electromagnetic Compatibility (EMC), 2010 IEEE International Symposium on*, 2010, pp. 132–137.

[71] M. G. Kuhn, "Compromising emanations of lcd tv sets," *IEEE Transactions on Electromagnetic Compatibility*, pp. 564–570, 2013.

[72] T. Plos, M. Hutter, and C. Herbst, "Enhancing side-channel analysis with low-cost shielding techniques," in *Proceedings of Austrochip*, 2008.

[73] F. Poucheret, L. Barthe, P. Benoit, L. Torres, P. Maurine, and M. Robert, "Spatial EM jamming: A countermeasure against EM Analysis?" In *Proceedings of the 18th IEEE/IFIP VLSI System on Chip Conference (VLSI-SoC)*, 2010, pp. 105–110.

[74] H. Tanaka, "Information leakage via electromagnetic emanations and evaluation of Tempest countermeasures," in *Lecture notes in computer science, Springer*, 2007, pp. 167–179.

[75] Y. ichi Hayashi, N. Homma, T. Mizuki, H. Shimada, T. Aoki, H. Sone, L. Sauvage, and J.-L. Danger, "Efficient evaluation of em radiation associated with information leakage from cryptographic devices," *IEEE Transactions on Electromagnetic Compatibility*, vol. 55, no. 3, pp. 555–563, 2013.

[76] H. Sekiguchi and S. Seto, "Study on maximum receivable distance for radiated emission of information technology equipment causing information leakage," *IEEE Transactions on Electromagnetic Compatibility*, vol. 55, no. 3, pp. 547–554, 2013.

[77] Y. ichi Hayashi, N. Homma, T. Mizuki, T. Aoki, H. Sone, L. Sauvage, and J.-L. Danger, "Analysis of electromagnetic information leakage from cryptographic devices with different physical structures," *IEEE Transactions on Electromagnetic Compatibility*, vol. 55, no. 3, pp. 571–580, 2013.

[78] D. Genkin, I. Pipman, and E. Tromer, "Get your hands of my laptop: Physical side-channel key-extraction attacks on pcs," in *Proceedings of Cryptographic Hardware and Embedded Systems*, ser. CHES '14', 2014.

[79] R. Callan, A. Zajić, and M. Prvulovic, "A practical methodology for measuring the side-channel signal available to the attacker for instruction-level events," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47, 2014.

[80] B. Durak, *Controlled CPU TEMPEST Emanations*, `http://cryptome.org/tempest-cpu.htm`, accessed April 2, 2016.

[81] S. Clark, H. Mustafa, B. Ransford, J. Sorber, K. Fu, and W. Xu, "Current events: Identifying webpages by tapping the electrical outlet," in *Computer Security- ESORICS 2013*, ser. Lecture Notes in Computer Science, J. Crampton, S. Jajodia, and K. Mayes, Eds., vol. 8134, Springer Berlin Heidelberg, 2013, pp. 700–717, ISBN: 978-3-642-40202-9.

[82] C. R. A. González and J. H. Reed, "Power fingerprinting in sdr integrity assessment for security and regulatory compliance," *Analog Integrated Circuits and Signal Processing*, vol. 69, no. 2-3, pp. 307–327, 2011.

[83] S. S. Clark, B. Ransford, A. Rahmati, S. Guineau, J. Sorber, W. Xu, and K. Fu, "Wattsupdoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices," in *Presented as part of the 2013 USENIX Workshop on Health Information Technologies*, 2013.

[84] R. Callan, N. Basta, A. Zajic, and M. Prvulovic, "A new approach for measuring electromagnetic side-channel energy available to the attacker in modern processor-memory systems," in *Proceedings of the 9th European Conference on Antennas and Propagation (EuCAP)*, 2015.

[85] R. Callan, A. Zajic, and M. Prvulovic, "FASE: Finding Amplitude-modulated Side-channel Emanations," in *the 42nd International Symposium on Computer Architecture (ISCA)*, 2015.

[86] R. Callan, N. Popovic, A. Daruna, E. Pollmann, A. Zajic, and M. Prvulovic, "Comparison of electromagnetic side-channel energy available to the attacker from different computer systems," in *Proceedings of the 2015 IEEE International Symposium on Electromagnetic Compatibility (EMC)*, 2015.

[87] M. R. Guthaus, J. S. Pingenberg, D. Emst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, ser. WWC '01, 2001.

[88] E. Sejdić, I. Djurović, and J. Jiang, "Time–frequency feature representation using energy concentration: An overview of recent advances," *Digit. Signal Process.*, vol. 19, no. 1, pp. 153–183, Jan. 2009.

[89] AARONIA, *Datasheet: Rf near field probe set dc to 9ghz*, `http://www.aaronia.com/Datasheets/Antennas/RF-Near-Field-Probe-Set.pdf`, accessed April 6, 2016.

[90] ARM, *Arm performance monitor unit*, `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0388f/Bcgddibf.html`, accessed April 2, 2016.

[91] G. Paoloni, "White paper: How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures," Intel Corporation, Tech. Rep., 2010.

[92] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, *SESC simulator*, http://sesc.sourceforge.net, 2005.

[93] G. Reinman and N. Jouppi, "Cacti 2.0: An integrated cache timing and power model," *Technical Report*, 2000.

[94] D. Brooks, V. Tiwari, and M. Martonosi, in *ACM/IEEE International Symposium on Computer Architecture*, ser. ISCA-27, 2000, pp. 83–94.

[95] D. I. F. Gregory W Corder, *Nonparametric Statistics for Non-Statisticians: A Step-by-Step Approach*. Wiley, 2011, ISBN: 9781118211250.

[96] F. J. Massey Jr, "The kolmogorov-smirnov test for goodness of fit," *Journal of the American statistical Association*, vol. 46, no. 253, pp. 68–78, 1951.

[97] ARM, *Arm cortex a8 processor manual*, https://www.arm.com/products/processors/cortex-a/cortex-a8.php, accessed April 3, 2016.

[98] C. Schmidt, *Low Level Virtual Machine (LLVM)*, Retrieved on April 1 from https://github.com/llvm-mirror/llvm, 2014.

[99] *Lewansoul learm 6dof full metal robotic arm*, Retrieved on April 2019 from https://www.amazon.com/LewanSoul-Controller-Wireless-Software-Tutorials/dp/B074T6DPKX.

[100] *Rtl-sdr*, https://www.rtl-sdr.com/rtl-sdr-quick-start-guide/, accessed April 2019).

[101] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07, Alexandria, Virginia, USA: ACM, 2007, pp. 552–561, ISBN: 978-1-59593-703-2.

[102] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11, Hong Kong, China: ACM, 2011, pp. 30–40, ISBN: 978-1-4503-0564-8.

[103] B. Wijnen, E. J. Hunt, G. C. Anzalone, and J. M. Pearce, "Open-source syringe pump library," *PLOS ONE*, vol. 9, no. 9, pp. 1–8, Sep. 2014.

[104] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-flat: Control-flow attestation for embedded systems software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, Vienna, Austria: ACM, 2016, pp. 743–754, ISBN: 978-1-4503-4139-4.

[105] N. Sehatbakhsh, M. Alam, A. Nazari, A. Zajic, and M. Prvulovic, "Syndrome: Spectral analysis for anomaly detection on medical iot and embedded devices," in *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2018, pp. 1–8.

[106] J. Salwan and A. Wirth, *Ropgadget: Gadgets finder for multiple architectures*, https://github.com/JonathanSalwan/ROPgadget, 2011 (accessed Feb. 1, 2018).

[107] *Horn antenna datasheet*, https://www.com-power.com/ah118hornantenna.html, 2015 (accessed Nov. 5, 2017).

[108] S. A. Carr and M. Payer, "Datashield: Configurable data confidentiality and integrity," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17, Abu Dhabi, United Arab Emirates: ACM, 2017, pp. 193–204, ISBN: 978-1-4503-4944-4.

[109] D. Quarta, M. Pogliani, M. Polino, F. Maggi, A. M. Zanchettin, and S. Zanero, "An experimental security analysis of an industrial robot controller," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 268–286.

[110] *Arduino servo library*, https://www.arduino.cc/en/Reference/Servo, accessed April 2019).

[111] Ubuntu, *Lightdm*, https://wiki.ubuntu.com/LightDM, 2017 (accessed Feb. 1, 2018).

[112] D. Genkin, L. Pachmanov, I. Pipman, and E. Tromer, "Stealing keys from pcs using a radio: Cheap electromagnetic attacks on windowed exponentiation," in *Proceedings of the 17th International Workshop on Cryptographic Hardware and Embedded Systems — CHES 2015*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 207–228, ISBN: 978-3-662-48324-4.

[113] Y.-i. Hayashi, N. Homma, T. Mizuki, H. Shimada, T. Aoki, H. Sone, L. Sauvage, and J.-L. Danger, "Efficient evaluation of em radiation associated with information leakage from cryptographic devices," *IEEE Transactions on Electromagnetic Compatibility*, vol. 55, no. 3, pp. 555–563, 2013.

[114] D. Genkin, A. Shamir, and E. Tromer, "Rsa key extraction via low-bandwidth acoustic cryptanalysis," in *Advances in Cryptology – CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 444–461, ISBN: 978-3-662-44371-2.

[115] U. Rührmair, X. Xu, J. Sölter, A. Mahmoud, M. Majzoobi, F. Koushanfar, and W. P. Burleson, "Efficient power and timing side channels for physical unclonable functions," in *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, 2014, pp. 476–492.

[116] R. Callan, A. G. Zajic, and M. Prvulovic, "A practical methodology for measuring the side-channel signal available to the attacker for instruction-level events," in *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*, 2014, pp. 242–254.

[117] ——, "FASE: finding amplitude-modulated side-channel emanations," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, 2015.

[118] J.-J. Quisquater and D. Samyde, "Automatic code recognition for smart cards using a kohonen neural network," in *Proceedings of the 5th Conference on Smart Card Research and Advanced Application Conference - Volume 5*, ser. CARDIS'02, San Jose, CA: USENIX Association, 2002, pp. 6–6.

[119] T. Eisenbarth, C. Paar, and B. Weghenkel, "Building a side channel based disassembler," in *Transactions on Computational Science X*, M. L. Gavrilova, C. J. K. Tan, and E. D. Moreno, Eds., Berlin, Heidelberg: Springer-Verlag, 2010, pp. 78–99, ISBN: 3-642-17498-1, 978-3-642-17498-8.

[120] M. Msgna, K. Markantonakis, and K. Mayes, "Precise instruction-level side channel profiling of embedded processors," in *Proceedings of the 10th International Conference on Information Security Practice and Experience - Volume 8434*, ser. ISPEC 2014, Fuzhou, China: Springer-Verlag New York, Inc., 2014, pp. 129–143, ISBN: 978-3-319-06319-5.

[121] D. Strobel, F. Bache, D. Oswald, F. Schellenberg, and C. Paar, "Scandalee: A side-channel-based disassembler using local electromagnetic emanations," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2015, pp. 139–144.

[122] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP '15, Washington, DC, USA: IEEE Computer Society, 2015, pp. 605–622, ISBN: 978-1-4673-6949-7.

[123] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14, San Diego, CA: USENIX Association, 2014, pp. 719–732, ISBN: 978-1-931971-15-7.

[124] T. Kim, M. Peinado, and G. Mainar-Ruiz, "Stealthmem: System-level protection against cache-based side channel attacks in the cloud," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security'12, Bellevue, WA: USENIX Association, 2012, pp. 11–11.

[125] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07, San Diego, California, USA: ACM, 2007, pp. 494–505, ISBN: 978-1-59593-706-3.

[126] R. Callan, F. Behrang, A. G. Zajic, M. Prvulovic, and A. Orso, "Zero-overhead profiling via EM emanations," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, 2016, pp. 401–412.

[127] N. Sehatbakhsh, A. Nazari, A. Zajic, and M. Prvulovic, "Spectral profiling: Observer-effect-free profiling by monitoring em emanations," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–11.

[128] Y. Han, S. Etigowni, H. Liu, S. Zonouz, and A. Petropulu, "Watch me, but don't touch me! contactless control flow monitoring via electromagnetic emanations," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, Dallas, Texas, USA: ACM, 2017, pp. 1095–1108, ISBN: 978-1-4503-4946-8.

[129] H. A. Khan, N. Sehatbakhsh, L. N. Nguyen, R. L. Callan, A. Yeredor, M. Prvulovic, and A. Zajic, "IDEA: Intrusion detection through electromagnetic-signal analysis for critical embedded and cyber-physical systems," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2019.

[130] L. J. Mariano, A. Aubuchon, T. Lau, O. Ozdemir, T. Lazovich, and J. Coakley, "Classification of electronic devices and software processes via unintentional electronic emissions with neural decoding algorithms," *IEEE Transactions on Electromagnetic Compatibility*, pp. 1–8, 2019.

[131] Z. Hadjilambrou, S. Das, M. A. Antoniades, and Y. Sazeides, "Leveraging CPU electromagnetic emanations for voltage noise characterization," in *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 573–585.

[132] H. Kim, J. Smith, and K. G. Shin, "Detecting energy-greedy anomalies and mobile malware variants," in *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '08, Breckenridge, CO, USA: ACM, 2008, pp. 239–252, ISBN: 978-1-60558-139-2.

[133] L. Liu, G. Yan, X. Zhang, and S. Chen, "Virusmeter: Preventing your cellphone from spies," in *Recent Advances in Intrusion Detection, 12th International Symposium, RAID 2009, Saint-Malo, France, September 23-25, 2009. Proceedings*, 2009, pp. 244–264.

[134] C. R. Aguayo González and J. H. Reed, "Power fingerprinting in sdr integrity assessment for security and regulatory compliance," *Analog Integr. Circuits Signal Process.*, vol. 69, no. 2-3, pp. 307–327, Dec. 2011.

[135] B. Chen, X. Dong, G. Bai, S. Jauhar, and Y. Cheng, "Secure and efficient software-based attestation for industrial control devices with arm processors," in *Proceedings of the 33rd Annual Computer Security Applications Conference*, ser. ACSAC 2017, Orlando, FL, USA: ACM, 2017, pp. 425–436, ISBN: 978-1-4503-5345-8.

[136] R. W. Gardner, S. Garera, and A. D. Rubin, "Detecting code alteration by creating a temporary memory bottleneck," *IEEE Transactions on Information Forensics and Security*, vol. 4, no. 4, pp. 638–650, 2009.

[137] Y. Li, J. M. McCune, and A. Perrig, "Viper: Verifying the integrity of peripherals' firmware," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11, Chicago, Illinois, USA: ACM, 2011, pp. 3–16, ISBN: 978-1-4503-0948-6.

[138] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, "Swatt: Software-based attestation for embedded devices," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2004, pp. 272–282.

[139] A. Seshadri, M. Luk, and A. Perrig, "Sake: Software attestation for key establishment in sensor networks," *Ad Hoc Netw.*, vol. 9, no. 6, pp. 1059–1067, Aug. 2011.

[140] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla, "Scuba: Secure code update by attestation in sensor networks," in *Proceedings of the 5th ACM Workshop on Wireless Security*, ser. WiSe '06, Los Angeles, California: ACM, 2006, pp. 85–94, ISBN: 1-59593-557-6.

[141] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, "Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP '05, Brighton, United Kingdom: ACM, 2005, pp. 1–16, ISBN: 1-59593-079-5.

[142] A. Klimov and A. Shamir, "New cryptographic primitives based on multiword t-functions," in *Fast Software Encryption*, B. Roy and W. Meier, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1–15, ISBN: 978-3-540-25937-4.

[143] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "On the difficulty of software-based attestation of embedded devices," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09, Chicago, Illinois, USA: ACM, 2009, pp. 400–409, ISBN: 978-1-60558-894-0.

[144] F. Armknecht, A. Sadeghi, S. Schulz, and C. Wachsmann, "A security framework for the analysis and design of software attestation," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13, Berlin, Germany: ACM, 2013, pp. 1–12, ISBN: 978-1-4503-2477-9.

[145] L. Li, H. Hu, J. Sun, Y. Liu, and J. S. Dong, "Practical analysis framework for software-based attestation scheme," in *Formal Methods and Software Engineering*, S. Merz and J. Pang, Eds., Cham: Springer International Publishing, 2014, pp. 284–299, ISBN: 978-3-319-11737-9.

[146] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[147] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[148] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: Exposing the perils of security-oblivious energy management," in *Proceedings of the 26th USENIX Conference on Security Symposium*, 2017, pp. 1057–1074, ISBN: 978-1-931971-40-9.

[149] S. K. Khatamifard, L. Wang, A. Das, S. Kose, and U. R. Karpuzcu, "POWERT channels: A novel class of covert communication exploiting power management vulnerabilities," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 291–303.

[150] M. Guri, A. Kachlon, O. Hasson, G. Kedma, Y. Mirsky, and Y. Elovici, "GSMem: Data exfiltration from air-gapped computers over GSM frequencies," in *24th USENIX Security Symposium*, 2015, pp. 849–864, ISBN: 978-1-931971-232.

[151] M. Guri, M. Monitz, and Y. Elovici, "USBee: Air-gap covert-channel via electromagnetic emission from USB," *CoRR*, vol. abs/1608.08397, 2016. arXiv: 1608.08397.

[152] M. Hanspach and M. Goetz, "On covert acoustical mesh networks in air," *Journal of Communications*, vol. 8, no. 11, 2013.

[153] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced CPU energy," in *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 1994.

[154] Intel-Corp., "Enhanced Intel® SpeedStep® Technology for the Intel® Pentium® M Processor," Tech. Rep., 2004.

[155] J. M. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits (2nd Edition)*. Prentice Hall, 2013, ISBN: 978-0130909961.

[156] Intel-Corp., "Power Management in Intel® Architecture Servers," Tech. Rep., 2009.

[157] ——, "Intel® Xeon® Processor E3-1200 v3 Family," Tech. Rep., 2014.

[158] H. Mujtaba, *Intel's 6th Gen Skylake Unwrapped – CPU Microarchitecture, Gen9 Graphics Core and Speed Shift Hardware P-State*, https://wccftech.com/idf15-intel-skylake-analysis-cpu-gpu-microarchitecture-ddr4-memory-impact/4/, 2015 (accessed Feb., 2019).

[159] Intel-Corp., "Voltage Regulator-Down 11.1: Processor Power Delivery Design Guide," Tech. Rep., 2009.

[160] ——, "ER6230QI: 3A DC-DC Step-Down Switching Regulator," Tech. Rep., 2018.

[161] J. Sun, M. Xu, Y. Ren, and F. C. Lee, "Light-load efficiency improvement for buck voltage regulators," *IEEE Transactions on Power Electronics*, vol. 24, no. 3, pp. 742–751, 2009.

[162] J. Su and C. Liu, "A novel phase-shedding control scheme for improved light load efficiency of multiphase interleaved DC–DC converters," *IEEE Transactions on Power Electronics*, vol. 28, no. 10, pp. 4742–4752, 2013.

[163] Y. Ahn, I. Jeon, and J. Roh, "A multiphase buck converter with a rotating phase-shedding scheme for efficient light-load control," *IEEE Journal of Solid-State Circuits*, vol. 49, no. 11, pp. 2673–2683, 2014.

[164] E. Cole, *Advanced persistent threat: understanding the danger and how to protect your organization*. Newnes, 2012.

[165] RTL-SDR.COM, *RTL-SDR Blog V3 Users*, https://www.rtl-sdr.com/rtl-sdr-quick-start-guide/, 2019 (accessed Feb., 2019).

[166] Universal-Radio-Inc., *AOR LA390 Wideband Loop Antenna*, https://www.universal-radio.com/catalog/sw_ant/2320.html, 2014 (accessed Feb., 2019).

[167] J. V. Monaco, "SoK: Keylogging side channels," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 211–228.

[168] M. Schwarz, M. Lipp, D. Gruss, S. Weiser, C. Maurice, R. Spreitzer, and S. Mangard, "Keydrown: Eliminating software-based keystroke timing side-channel attacks," in *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2018.

[169] D. Wang, A. Neupane, Z. Qian, N. B. Abu-Ghazaleh, S. V. Krishnamurthy, E. J. Colbert, and P. Yu, "Unveiling your keystrokes: A cache-based side-channel attack

on graphics libraries.," in *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2019.

[170] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *Proceedings of the 24th USENIX Conference on Security Symposium*, 2015, pp. 897–912, ISBN: 978-1-931971-232.

[171] P. Vila and B. Kopf, "Loophole: Timing attacks on shared event loops in chrome," in *26th USENIX Security Symposium*, 2017, pp. 849–864, ISBN: 978-1-931971-40-9.

[172] K. Ali, A. X. Liu, W. Wang, and M. Shahzad, "Keystroke recognition using WiFi signals," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2015, pp. 90–102, ISBN: 978-1-4503-3619-2.

[173] M. Vuagnoux and S. Pasini, "Compromising electromagnetic emanations of wired and wireless keyboards," in *Proceedings of the 18th Conference on USENIX Security Symposium*, 2009, pp. 1–16.

[174] Y. Berger, A. Wool, and A. Yeredor, "Dictionary attacks using keyboard acoustic emanations," in *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, 2006, pp. 245–254, ISBN: 1-59593-518-5.

[175] X. Liu, Z. Zhou, W. Diao, Z. Li, and K. Zhang, "When good becomes evil: Keystroke inference with smartwatch," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015, pp. 1273–1285, ISBN: 978-1-4503-3832-5.

[176] D. Asonov and R. Agrawal, "Keyboard acoustic emanations," in *IEEE Symposium on Security and Privacy (SP).*, 2004, pp. 3–11.

[177] T. A. Salthouse, "Perceptual, cognitive, and motoric aspects of transcription typing.," *Psychological bulletin*, vol. 99, no. 3, p. 303, 1986.

[178] A. M. Feit, D. Weir, and A. Oulasvirta, "How we type: Movement strategies and performance in everyday typing," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 2016, pp. 4262–4273, ISBN: 978-1-4503-3362-7.

[179] C. H. Gebotys, S. Ho, and C. C. Tiu, "EM analysis of Rijndael and ECC on a wireless java-based PDA," in *Proceedings of the 7th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, Edinburgh, UK, 2005, pp. 250–264, ISBN: 3-540-28474-5, 978-3-540-28474-1.

[180] G. Camurati, S. Poeplau, M. Muench, T. Hayes, and A. Francillon, "Screaming channels: When electromagnetic side channels meet radio transceivers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018, pp. 163–177, ISBN: 978-1-4503-5693-0.

[181] H. J. Highland, "Random bits & bytes: Electromagnetic radiation revisited," *Comput. Secur.*, vol. 5, no. 2, pp. 85–93, Jun. 1986.

[182] S. S. Clark, H. Mustafa, B. Ransford, J. Sorber, K. Fu, and W. Xu, "Current events: Identifying webpages by tapping the electrical outlet," in *Computer Security (ESORICS)*, 2013, pp. 700–717, ISBN: 978-3-642-40203-6.

[183] R. Callan, F. Behrang, A. Zajic, M. Prvulovic, and A. Orso, "Zero-overhead profiling via EM emanations," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*, 2016, 401–412, ISBN: 9781450343909.

[184] S. S. Clark, B. Ransford, A. Rahmati, S. Guineau, J. Sorber, W. Xu, and K. Fu, "Wattsupdoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices," in *Presented as part of the 2013 USENIX Workshop on Health Information Technologies*, 2013.

[185] N. Sehatbakhsh, A. Nazari, M. Alam, F. Werner, Y. Zhu, A. Zajic, and M. Prvulovic, "REMOTE: Robust external malware detection framework by using electromagnetic signals," *IEEE Transactions on Computers*, pp. 1–1, 2019.

[186] N. Sehatbakhsh, A. Nazari, H. Khan, A. Zajic, and M. Prvulovic, "EMMA: Hardware/software attestation framework for embedded systems using electromagnetic signals," in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 983–995, ISBN: 978-1-4503-6938-1.

[187] R. JayashankaraShridevi, C. Rajamanikkam, K. Chakraborty, and S. Roy, "Catching the flu: Emerging threats from a third party power management unit," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[188] C. Tokunaga and D. Blaauw, "Secure aes engine with a local switched-capacitor current equalizer," in *2009 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, 2009, 64–65,65a.

[189] D. Das, S. Maity, S. B. Nasir, S. Ghosh, A. Raychowdhury, and S. Sen, "ASNI: Attenuated signature noise injection for low-overhead power side-channel attack immunity," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 10, pp. 3300–3311, 2018.

[190] O. A. Uzun and S. Köse, "Converter-gating: A power efficient and secure on-chip power delivery system," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 4, no. 2, pp. 169–179, 2014.

[191] M. Kar, A. Singh, S. Mathew, S. Ghosh, A. Rajan, V. De, R. A. Beyah, and S. Mukhopadhyay, "Blindsight: Blinding EM side-channel leakage using built-in fully integrated inductive voltage regulator," *CoRR*, vol. abs/1802.09096, 2018. arXiv: 1802.09096.

[192] M. Kar, A. Singh, S. K. Mathew, A. Rajan, V. De, and S. Mukhopadhyay, "Reducing power side-channel information leakage of aes engines using fully integrated inductive voltage regulator," *IEEE Journal of Solid-State Circuits*, vol. 53, no. 8, pp. 2399–2414, 2018.

[193] A. Singh, M. Kar, S. Mathew, A. Rajan, V. De, and S. Mukhopadhyay, "Improved power side channel attack resistance of a 128-bit AES engine with random fast voltage dithering," in *ESSCIRC 2017 - 43rd IEEE European Solid State Circuits Conference*, 2017, pp. 51–54.

[194] A. Althoff, J. McMahan, L. Vega, S. Davidson, T. Sherwood, M. B. Taylor, and R. Kastner, "Hiding intermittent information leakage with architectural support for blinking," in *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 638–649, ISBN: 978-1-5386-5984-7.

[195] J. Balasch, B. Gierlichs, O. Reparaz, and I. Verbauwhede, "DPA, bitslicing and masking at 1 GHz," *IACR Cryptology ePrint Archive*, vol. 2015, p. 727, 2015.

[196] D. J. Bernstein, J. Breitner, D. Genkin, L. Groot Bruinderink, N. Heninger, T. Lange, C. van Vredendaal, and Y. Yarom, "Sliding right into disaster: Left-to-right sliding windows leak," in *Cryptographic Hardware and Embedded Systems (CHES)"*, 2017, pp. 555–576, ISBN: 978-3-319-66787-4.

[197] J. Brouchier, T. Kean, C. Marsh, and D. Naccache, "Temperature attacks," *IEEE Security Privacy*, vol. 7, no. 2, pp. 79–82, 2009.

[198] S. Chari, J. R. Rao, and P. Rohatgi, "Template attacks," in *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2003, pp. 13–28, ISBN: 3-540-00409-2.

[199] B. B. Yilmaz, N. Sehatbakhsh, A. Zajić, and M. Prvulovic, "Communication model and capacity limits of covert channels created by software activities," *IEEE Transactions on Information Forensics and Security*, pp. 1–1, 2019.

[200] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore

and manycore architectures," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 469–480, ISBN: 978-1-60558-798-1.

[201] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 2011, pp. 694–701, ISBN: 978-1-4577-1398-9.

[202] S. J. E. Wilton and N. P. Jouppi, "CACTI: An enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, 1996.

[203] E. K. Ardestani and J. Renau, "ESESC: A fast multicore simulator using time-based sampling," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 448–459, ISBN: 978-1-4673-5585-8.

[204] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSS: A full system simulator for multicore x86 cpus," in *48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2011, pp. 1050–1055.

[205] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.

[206] M. Taram, A. Venkat, and D. Tullsen, "Mobilizing the Micro-Ops: Exploiting context sensitive decoding for security and energy efficiency," in *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 624–637.

[207] M. Andrysco, A. Nötzli, F. Brown, R. Jhala, and D. Stefan, "Towards verified, constant-time floating point operations," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018, pp. 1369–1382, ISBN: 978-1-4503-5693-0.

[208] J. Yu, L. Hsiung, M. E. Hajj, and C. W. Fletcher, "Data oblivious ISA extensions for side channel-resistant and high performance computing," in *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, `https://eprint.iacr.org/2018/808`, 2019.

[209] A. Rane, C. Lin, and M. Tiwari, "Secure, precise, and fast floating-point operations on x86 processors," in *Proceedings of the 25th USENIX Security Conference*, 2016, pp. 71–86, ISBN: 978-1-931971-32-4.

[210] K. Nayak, C. W. Fletcher, L. Ren, N. Chandran, S. V. Lokam, E. Shi, and V. Goyal, "HOP: Hardware makes obfuscation practical.," in *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[211] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi, "Ghostrider: A hardware-software system for memory trace oblivious computation," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015, pp. 87–101, ISBN: 978-1-4503-2835-7.

[212] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *Proceedings of the 24th USENIX Security Conference*, 2015, pp. 431–446, ISBN: 978-1-931971-232.

[213] D. I. Gorman, M. R. Guthaus, and J. Renau, "Architectural opportunities for novel dynamic EMI shifting (DEMIS)," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 774–785, ISBN: 978-1-4503-4952-9.

[214] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, "MicroWalk: A framework for finding side channels in binaries," in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018, pp. 161–173, ISBN: 978-1-4503-6569-7.

[215] J. Chen, Y. Feng, and I. Dillig, "Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 875–890, ISBN: 978-1-4503-4946-8.

[216] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2018, pp. 15–26, ISBN: 978-1-4503-5699-2.

[217] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013, pp. 559–570, ISBN: 978-1-4503-2079-5.

[218] R. Callan, N. Popovic, A. Daruna, E. Pollmann, A. Zajic, and M. Prvulovic, "Comparison of electromagnetic side-channel energy available to the attacker from different computer systems," in *IEEE International Symposium on Electromagnetic Compatibility (EMC)*, 2015, pp. 219–223.

[219] R. Callan, N. Popovic, A. Zajic, and M. Prvulovic, "A new approach for measuring electromagnetic side-channel energy available to the attacker in modern processor-memory systems," in *Proceedings of the 9th European Conference on Antennas and Propagation (EuCAP)*, 2015.

[220] B. Berkay Yilmaz, R. Callan, M. Prvulovic, and A. Zajić, "Quantifying information leakage in a processor caused by the execution of instructions," in *IEEE Military Communications Conference (MILCOM)*, 2017, pp. 255–260.

[221] D. McCann, E. Oswald, and C. Whitnall, "Towards practical tools for side channel aware software engineering: Grey box' modelling for instruction leakages," in *Proceedings of the 26th USENIX Security Conference*, 2017, pp. 199–216, ISBN: 978-1-931971-40-9.

[222] A. Barenghi and G. Pelosi, "Side-channel security of superscalar CPUs: Evaluating the impact of micro-architectural features," in *Proceedings of the 55th Annual Design Automation Conference (DAC)*, 2018, 120:1–120:6, ISBN: 978-1-4503-5700-5.

[223] B. B. Yilmaz, R. L. Callan, M. Prvulovic, and A. Zajić, "Capacity of the EM covert/side-channel created by the execution of instructions in a processor," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 3, pp. 605–620, 2017.

[224] A. Waterman and K. Asanovic, *RISC-V instruction reference*, `https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf`, 2019 (accessed Nov. 6, 2019).

[225] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, "The RISC-V instruction set manual. volume 1: User-level ISA, version 2.0," CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING and COMPUTER SCIENCES, Tech. Rep., 2014.

[226] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proceedings of the 24th Annual International Symposium on Microarchitecture (MICRO)*, 1991, pp. 51–61, ISBN: 0-89791-460-0.

[227] B. B. Yilmaz, E. M. Ugurlu, A. Zajic, and M. Prvulovic, "Instruction level program tracking using electromagnetic emanations," in *Proceedings of the SPIE*, International Society for Optics and Photonics, vol. 11011, 2019.

[228] A. V. Oppenheim, A. S. Willsky, and S. H. Nawab, *Signals &Amp; Systems (2Nd Ed.)* Prentice-Hall, Inc., 1996, ISBN: 0-13-814757-4.

[229] W. van Eck, "Electromagnetic radiation from video display units: An eavesdropping risk?" *Computers and Security*, vol. 4, no. 4, pp. 269 –286, 1985.

[230] D. J. Hand, "Statistical concepts: A second course, fourth edition by richard g. lomax, debbie l. hahs-vaughn," *International Statistical Review*, vol. 80, no. 3, pp. 491–491, 2012.

[231] Terassic, *DE0-CV*, http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=364, 2019 (accessed Nov. 6, 2019).

[232] Tektronix, *Tbs1000-digital-storage-oscilloscope*, https://www.tek.com/oscilloscope/tbs1000-digital-storage-oscilloscope, 2019 (accessed Nov. 6, 2019).

[233] W. B. Frakes and R. Baeza-Yates, *Information retrieval: Data structures & algorithms*. Prentice Hall Englewood Cliffs, NJ, 1992, vol. 331.

[234] Terassic, *DE1 evaluation board*, https://www.intel.com/content/www/us/en/programmable/solutions/partners/partner-profile/terasic-inc-/board/altera-de1-board.html, 2019 (accessed Nov. 6, 2019).

[235] Xilinx, *Digilent evalaution board*, https://store.digilentinc.com/arty-a7-artix-7-fpga-development-board-for-makers-and-hobbyists/, 2019 (accessed Nov. 6, 2019).

[236] G. Becker, J Cooper, E. DeMulder, G. Goodwill, J. Jaffe, G Kenworthy, T Kouzminov, A Leiserson, M Marson, and P. Rohatgi, "Test vector leakage assessment (TVLA) methodology in practice," in *International Cryptographic Module Conference*, 2013.

[237] G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi, *A testing methodology for side-channel resistance validation*, 2011.

[238] A. Heuser, W. Schindler, and M. Stöttinger, "Revealing side-channel issues of complex circuits by enhanced leakage models," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012, pp. 1179–1184.

[239] T. Ming, W. Pengbo, M. Xiaoqi, C. Wenjie, Z. Huanguo, P. Guojun, and J. Danger, "An efficient SCA leakage model construction method under predictable evaluation," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 12, pp. 3008–3018, 2018.

[240] S. Mangard, E. Oswald, and F. Standaert, "One for all - all for one: Unifying standard differential power analysis attacks," *IET Information Security*, vol. 5, no. 2, pp. 100–110, 2011.

[241] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 1999, pp. 388–397, ISBN: 3-540-66347-9.

[242] B. B. Yilmaz, M. Prvulovic, and A. Zajić, "Electromagnetic side channel information leakage created by execution of series of instructions in a computer processor," *IEEE Transactions on Information Forensics and Security*, pp. 1–1, 2019.

[243] R. Callan, A. Zajić, and M. Prvulovic, "FASE: Finding amplitude-modulated side-channel emanations," in *ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 592–603.

[244] M. Prvulovic, A. Zajić, R. L. Callan, and C. J. Wang, "A method for finding frequency-modulated and amplitude-modulated electromagnetic emanations in computer systems," *IEEE Transactions on Electromagnetic Compatibility*, vol. 59, no. 1, pp. 34–42, 2017.

[245] B. B. Yilmaz, M. Prvulovic, and A. Zajić, "Capacity of deliberate side-channels created by software activities," in *IEEE Military Communications Conference (MILCOM)*, 2018, pp. 237–242.

[246] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic, "EDDIE: EM-based detection of deviations in program execution," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 333–346, ISBN: 978-1-4503-4892-8.

[247] M. Taha and P. Schaumont, "Key updating for leakage resiliency with application to AES modes of operation," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 3, pp. 519–528, 2015.

[248] Z. He and R. B. Lee, "How secure is your cache against side-channel attacks?" In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 341–353, ISBN: 978-1-4503-4952-9.

[249] D. I. Gorman, R. T. Possignolo, and J. Renau, "EMI architectural model and core hopping," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 899–910, ISBN: 978-1-4503-6938-1.

[250] M. Dey, A. Nazari, A. Zajic, and M. Prvulovic, "EMPROF: Memory profiling via em-emanation in iot and hand-held devices," in *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 881–893.

[251] S. S. Clark, B. Ransford, A. Rahmati, S. Guineau, J. Sorber, K. Fu, and W. Xu, "WattsUpDoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices," in *Proceedings of the USENIX Conference on Safety, Security, Privacy and Interoperability of Health Information Technologies*, 2013, pp. 9–9.

[252] C. R. Aguayo González and J. H. Reed, "Power fingerprinting in SDR integrity assessment for security and regulatory compliance," *Analog Integr. Circuits Signal Process.*, vol. 69, no. 2-3, pp. 307–327, Dec. 2011.

[253] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: Hardware-assisted data-flow isolation," in *IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 1–17.

VITA

Nader Sehatbakhsh received his B.Sc. degree in Electrical Engineering from the University of Tehran in 2013 and his M.Sc. in Electrical Engineering from the Georgia Institute of Technology in 2016. Since 2014, he has been a Graduate Research Assistant with CompArch and Electromagnetic Measurements in Communications and Computing (EMC^2) Labs, pursuing the Ph.D. degree in the School of Computer Science, Georgia Institute of Technology focusing on Computer Architecture, Embedded System and Hardware Security. His work has received multiple recognition including the Best Paper Award in MIRCO'49 for his work on using EM side-channel signals for software profiling.